



FREE eBook

LEARNING Angular 2

Free unaffiliated eBook created from
Stack Overflow contributors.

#angular2

Table of Contents

About.....	1
Chapter 1: Getting started with Angular 2	2
Remarks.....	2
Versions.....	2
Examples.....	3
Install angular2 with angular-cli.....	3
Prerequisites:	3
To setup a new project	3
To add to an existing project	3
Running The Project Locally	4
Generating Components, Directives, Pipes and Services	4
Getting started with Angular 2 without angular-cli.....	6
Step 1.....	6
Step 2.....	6
Step 3.....	8
Step 5.....	9
Step 6.....	9
Step 7.....	10
Step 8.....	10
What now?.....	11
Keeping Visual Studios in sync with NPM and NODE Updates.....	11
Getting through that pesky company proxy.....	11
Getting started with Angular 2 with node.js/expressjs backend (http example included).....	12
Prerequisites	12
Roadmap	12
Step 1.....	12
Step2.....	13
Step3.....	14
Let's dive into Angular 4!.....	18

Chapter 2: Advanced Component Examples	23
Remarks.....	23
Examples.....	23
Image Picker with Preview.....	23
Filter out table values by the input.....	24
Chapter 3: Ahead-of-time (AOT) compilation with Angular 2	26
Examples.....	26
1. Install Angular 2 dependencies with compiler.....	26
2. Add `angularCompilerOptions` to your `tsconfig.json` file.....	26
3. Run ngc, the angular compiler.....	26
4. Modify `main.ts` file to use NgFactory and static platform browser.....	26
Why we need compilation, Flow of events compilation and example?.....	27
Using AoT Compilation with Angular CLI.....	28
Chapter 4: Angular 2 - Protractor	29
Examples.....	29
Testing Navbar routing with Protractor.....	29
Angular2 Protractor - Installation.....	30
Chapter 5: Angular - ForLoop	32
Syntax.....	32
Remarks.....	32
Examples.....	32
Angular 2 for-loop.....	32
NgFor - Markup For Loop.....	33
*ngFor in the Table Rows.....	33
*ngFor with component.....	33
*ngFor X amount of items per row.....	34
Chapter 6: Angular 2 Change detection and manual triggering	35
Examples.....	35
Basic example.....	35
Chapter 7: Angular 2 Data Driven Forms	37
Remarks.....	37
Examples.....	37

Data driven form	37
Chapter 8: Angular 2 Forms Update	40
Remarks	40
Examples	40
Simple Password Change Form with Multi Control Validation	40
pw-change.template.html	40
pw-change.component.ts	41
pw-validators.ts	41
Angular 2 : Template Driven Forms	42
Angular 2 Form - Custom Email/Password Validation	43
Angular 2: Reactive Forms (a.k.a Model-driven Forms)	44
registration-form.component.ts	44
registration-form.html	44
Angular 2 Forms (Reactive Forms) with registration form and confirm password validation	45
app.module.ts	45
app.component.ts	45
app.component.html	46
validators.ts	46
Angular2 - Form Builder	47
Chapter 9: Angular material design	49
Examples	49
Md2Select	49
Md2Tooltip	49
Md2Toast	49
Md2Datepicker	50
Md2Accordion and Md2Collapse	50
Chapter 10: angular redux	51
Examples	51
Basic	51
Get current state	52
change state	52
Add redux chrome tool	53

Chapter 11: Angular RXJS Subjects and Observables with API requests	54
Remarks	54
Examples	54
Basic request	54
Encapsulating API requests	54
Wait for multiple requests	55
Chapter 12: Angular2 Animations	57
Introduction	57
Examples	57
Basic Animation - Transitions an element between two states driven by a model attribute	57
Chapter 13: Angular2 CanActivate	59
Examples	59
Angular2 CanActivate	59
Chapter 14: Angular2 Custom Validations	60
Parameters	60
Examples	60
Custom validator examples:	60
Using validators in the FormBuilder	60
get/set FormBuilder controls parameters	61
Chapter 15: Angular2 Databinding	62
Examples	62
@Input()	62
Parent Component : Initialize users lists	62
Chapter 16: Angular2 In Memory Web API	64
Remarks	64
Examples	64
Basic Setup	64
Setting Up Multiple Test API Routes	65
Chapter 17: Angular2 Input() output()	67
Examples	67
Input()	67

Parent Component : Initialize users lists.....	67
Simple example of Input Properties.....	68
Chapter 18: Angular2 provide external data to App before bootstrap.....	69
Introduction.....	69
Examples.....	69
Via Dependency Injection.....	69
Chapter 19: Angular2 using webpack.....	70
Examples.....	70
Angular 2 webpack setup.....	70
Chapter 20: Angular-cli.....	74
Introduction.....	74
Examples.....	74
Create empty Angular2 application with angular-cli.....	74
Generating Components, Directives, Pipes and Services.....	74
Adding 3rd party libs.....	74
build with angular-cli.....	75
New project with scss/sass as stylesheet.....	75
Set yarn as default package manager for @angular/cli.....	75
Requirements.....	76
Chapter 21: angular-cli test coverage.....	77
Introduction.....	77
Examples.....	77
A simple angular-cli command base test coverage.....	77
Detailed individual component base graphical test coverage reporting.....	77
Chapter 22: Animation.....	79
Examples.....	79
Transition between null states.....	79
Animating between multiple states.....	79
Chapter 23: Attribute directives to affect the value of properties on the host node by usi.....	81
Examples.....	81
@HostBinding.....	81

Chapter 24: Barrel	82
Introduction.....	82
Examples.....	82
Using Barrel.....	82
Chapter 25: Bootstrap Empty module in angular 2	83
Examples.....	83
An empty module.....	83
A module with networking on the web browser.....	83
Bootstrapping your module.....	83
Application Root Module.....	84
Static bootstrapping with factory classes.....	84
Chapter 26: Brute Force Upgrading	85
Introduction.....	85
Remarks.....	85
Examples.....	85
Scaffolding a New Angular CLI Project.....	85
Chapter 27: Bypassing Sanitizing for trusted values	86
Parameters.....	86
Remarks.....	86
SUPER IMPORTANT!	86
DISABLING SANITIZING LEAVES YOU AT RISK OF XSS (Cross-Site Scripting) AND OTHER ATTACK VEC.....	86
Examples.....	86
Bypassing Sanitizing with pipes (for code re-use).....	86
Chapter 28: Commonly built-in directives and services	90
Introduction.....	90
Examples.....	90
Location Class.....	90
AsyncPipe.....	90
Displaying current angular2 version used in your project.....	91
Currency Pipe.....	91
Chapter 29: Component interactions	93

Syntax.....	93
Parameters.....	93
Examples.....	93
Parent - Child interaction using @Input & @Output properties.....	93
Parent - Child interaction using ViewChild.....	94
Bidirectional parent-child interaction through a service.....	95
Chapter 30: Component interactions.....	98
Introduction.....	98
Examples.....	98
Pass data from parent to child with input binding.....	98
Chapter 31: Components.....	105
Introduction.....	105
Examples.....	105
A simple component.....	105
Templates & Styles.....	105
Passing template as a file path.....	105
Passing a template as an inline code.....	106
Passing an array of file paths.....	106
Passing an array of inline codes.....	106
Testing a Component.....	106
Nesting components.....	108
Chapter 32: Configuring ASP.net Core application to work with Angular 2 and TypeScript.....	109
Introduction.....	109
Examples.....	109
Asp.Net Core + Angular2 + Gulp.....	109
[Seed] Asp.Net Core + Angular2 + Gulp on Visual Studio 2017.....	113
MVC <-> Angular 2.....	113
Chapter 33: Create an Angular 2+ NPM package.....	115
Introduction.....	115
Examples.....	115
Simplest package.....	115

Configuration files	115
.gitignore	115
.npmignore	115
gulpfile.js	115
index.d.ts	116
index.js	116
package.json	116
dist/tsconfig.json	117
src/angular-x-minimal-npm-package.component.ts	118
src/angular-x-minimal-npm-package.component.html	118
src/angular-x-data-table.component.css	118
src/angular-x-minimal-npm-package.module.ts	118
Build and compile	118
Publish	119
Chapter 34: Creating an Angular npm library	120
Introduction	120
Examples	120
Minimal module with service class	120
File structure	120
Service and module	120
Compilation	121
NPM settings	122
Continuous integration	123
Chapter 35: CRUD in Angular2 with Restful API	125
Syntax	125
Examples	125
Read from an Restful API in Angular2	125
Chapter 36: custom ngx-bootstrap datepicker + input	127
Examples	127
custom ngx-bootstrap datepicker	127

Chapter 37: Debugging Angular2 typescript application using Visual Studio Code	130
Examples.....	130
Launch.json setup for you workspace.....	130
Chapter 38: Detecting resize events	132
Examples.....	132
A component listening in on the window resize event.....	132
Chapter 39: Directives	133
Syntax.....	133
Remarks.....	133
Examples.....	133
Attribute directive.....	133
Component is a directive with template.....	133
Structural directives.....	133
Custom directive.....	133
*ngFor.....	134
Copy to Clipboard directive.....	135
Testing a custom directive.....	136
Chapter 40: Directives & components : @Input @Output	139
Syntax.....	139
Examples.....	139
Input example.....	139
Angular2 @Input and @Output in a nested component.....	140
Angular2 @Input with asynchronous data.....	141
Parent component with async call to an endpoint	141
Child component which has async data as input	142
Chapter 41: Dropzone in Angular2	144
Examples.....	144
Dropzone.....	144
Chapter 42: Dynamically add components using ViewContainerRef.createComponent	146
Examples.....	146
A wrapper component that adds dynamic components declaratively.....	146

Dynamically add component on specific event(click).....	147
Rendered dynamically created component array on template html in Angular2.....	148
Chapter 43: EventEmitter Service.....	152
Examples.....	152
Class Overview.....	152
Class Component.....	152
Emmiting Events.....	152
Catching the event.....	152
Live example.....	153
Chapter 44: Example for routes such as /route/subroute for static urls.....	154
Examples.....	154
Basic route example with sub routes tree.....	154
Chapter 45: Feature Modules.....	155
Examples.....	155
A Feature Module.....	155
Chapter 46: How to use ngfor.....	156
Introduction.....	156
Examples.....	156
Unordered list example.....	156
More complex template example.....	156
Tracking current interaction example.....	156
Angular2 aliased exported values.....	156
*ngFor with pipe.....	157
Chapter 47: How to Use ngif.....	158
Introduction.....	158
Syntax.....	158
Examples.....	158
Display a loading message.....	158
Show Alert Message on a condition.....	159
To run a function at the start or end of *ngFor loop Using *ngIf.....	159
Use *ngIf with*ngFor.....	159
Chapter 48: Http Interceptor.....	161

Remarks.....	161
Examples.....	161
Simple Class Extending angular's Http class.....	161
Using our class instead of Angular's Http.....	162
Simple HttpClient AuthToken Interceptor (Angular 4.3+).....	163
Chapter 49: Installing 3rd party plugins with angular-cli@1.0.0-beta.10.....	164
Remarks.....	164
Examples.....	164
Adding jquery library in angular-cli project.....	164
Add 3rd party library that does not have typings.....	166
Chapter 50: Lazy loading a module.....	168
Examples.....	168
Lazy loading example.....	168
Chapter 51: Lifecycle Hooks.....	170
Remarks.....	170
Events availability.....	170
Events order.....	170
Further Reading.....	170
Examples.....	170
OnInit.....	170
OnDestroy.....	171
OnChanges.....	171
AfterContentInit.....	171
AfterContentChecked.....	172
AfterViewInit.....	172
AfterViewChecked.....	172
DoCheck.....	173
Chapter 52: Mocking @ngrx/Store.....	174
Introduction.....	174
Parameters.....	174
Remarks.....	174

Examples.....	174
Observer Mock.....	174
Unit Test For Component With Mock Store.....	175
Unit Test For Component Spying On Store.....	176
Angular 2 - Mock Observable (service + component).....	177
Simple Store.....	180
Chapter 53: Modules.....	183
Introduction.....	183
Examples.....	183
A simple module.....	183
Nesting modules.....	183
Chapter 54: ngrx.....	185
Introduction.....	185
Examples.....	185
Complete example : Login/logout a user.....	185
1) Define IUser interface.....	185
2) Declare the actions to manipulate the User.....	186
3) Define the initial state of the UserReducer.....	187
4) Create the reducer UserReducer.....	187
Reminder : A reducer needs to be initialized at some point.....	187
5) Import our UserReducer into our main module to build the Store.....	188
6) Use data from the Store to display information in our view.....	189
Chapter 55: Optimizing rendering using ChangeDetectionStrategy.....	192
Examples.....	192
Default vs OnPush.....	192
Chapter 56: OrderBy Pipe.....	194
Introduction.....	194
Examples.....	194
The Pipe.....	194
Chapter 57: Page title.....	197
Introduction.....	197

Syntax.....	197
Examples.....	197
changing the page title.....	197
Chapter 58: Pipes.....	198
Introduction.....	198
Parameters.....	198
Remarks.....	198
Examples.....	198
Chaining Pipes.....	198
Custom Pipes.....	198
Built-in Pipes.....	199
Angular2 comes with a few built-in pipes:.....	199
Example.....	200
hotel-reservation.component.ts.....	200
hotel-reservation.template.html.....	200
Output.....	200
Debugging With JsonPipe.....	200
Code.....	200
Output.....	201
Globally Available Custom Pipe.....	201
Creating Custom Pipe.....	201
Unwrap async values with async pipe.....	202
Extending an Existing Pipe.....	202
Stateful Pipes.....	203
Dynamic Pipe.....	204
Testing a pipe.....	206
Chapter 59: Routing.....	207
Examples.....	207
Basic Routing.....	207
Child Routes.....	209
ResolveData.....	210
Routing with Children.....	213

Chapter 60: Routing (3.0.0+)	215
Remarks.....	215
Examples.....	215
Bootstrapping.....	215
Configuring router-outlet.....	215
Changing routes (using templates & directives).....	216
Setting the Routes.....	217
Controlling Access to or from a Route.....	217
How Route Guards work	218
Route Guard Interfaces	218
Synchronous vs. Asynchronous Route Guards	218
Synchronous Route Guard.....	218
Asynchronous Route Guard.....	219
Add guard to route configuration.....	219
Use Guard in app bootstrap.....	220
Using Resolvers and Guards.....	220
Chapter 61: Service Worker	223
Introduction.....	223
Examples.....	223
Add Service Worker to our app.....	223
Chapter 62: Services and Dependency Injection	226
Examples.....	226
Example service.....	226
Example with Promise.resolve.....	227
Testing a Service.....	228
Chapter 63: Templates	231
Introduction.....	231
Examples.....	231
Angular 2 Templates.....	231
Chapter 64: Testing an Angular 2 App	233
Examples.....	233

Installing the Jasmine testing framework.....	233
Install.....	233
Verify.....	233
Setting up testing with Gulp, Webpack, Karma and Jasmine.....	233
Testing Http Service.....	238
Testing Angular Components - Basic.....	240
Chapter 65: Testing ngModel.....	242
Introduction.....	242
Examples.....	242
Basic test.....	242
Chapter 66: unit testing.....	244
Examples.....	244
Basic unit test.....	244
component file.....	244
Chapter 67: Update typings.....	246
Examples.....	246
Update typings when: typings WARN deprecated.....	246
Chapter 68: Use native webcomponents in Angular 2.....	247
Remarks.....	247
Examples.....	247
Include custom elements schema in your module.....	247
Use your webcomponent in a template.....	247
Chapter 69: Using third party libraries like jQuery in Angular 2.....	248
Introduction.....	248
Examples.....	248
Configuration using angular-cli.....	248
NPM.....	248
Assets Folder.....	248
Note.....	248
Using jQuery in Angular 2.x components.....	248
Chapter 70: Zone.js.....	250

Examples.....	250
Getting reference to NgZone.....	250
Using NgZone to do multiple HTTP requests before showing the data.....	250
Credits.....	252

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [angular-2](#)

It is an unofficial and free Angular 2 ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Angular 2.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Angular 2

Remarks

This section provides an overview of how to install and configure Angular2+ for use in various environments and IDE's using tools like the community developed [angular-cli](#).

The previous version of Angular is [AngularJS](#) or also named Angular 1. See here the [documentation](#).

Versions

Version	Release Date
4.3.3	2017-08-02
4.3.2	2017-07-26
4.3.1	2017-07-19
4.3.0	2017-07-14
4.2.0	2017-06-08
4.1.0	2017-04-26
4.0.0	2017-03-23
2.3.0	2016-12-08
2.2.0	2016-11-14
2.1.0	2016-10-13
2.0.2	2016-10-05
2.0.1	2016-09-23
2.0.0	2016-09-14
2.0.0-rc.7	2016-09-13
2.0.0-rc.6	2016-08-31
2.0.0-rc.5	2016-08-09
2.0.0-rc.4	2016-06-30

Version	Release Date
2.0.0-rc.3	2016-06-21
2.0.0-rc.2	2016-06-15
2.0.0-rc.1	2016-05-03
2.0.0-rc.0	2016-05-02

Examples

Install angular2 with angular-cli

This example is a quick setup of Angular 2 and how to generate a quick example project.

Prerequisites:

- [Node.js v4](#) or greater.
- [npm v3](#) or greater or [yarn](#).

Open a terminal and run the commands one by one:

```
npm install -g @angular/cli
```

or

```
yarn global add @angular/cli
```

depending on your choice of package manager.

The previous command installs **@angular/cli** globally, adding the executable `ng` to PATH.

To setup a new project

Navigate with the terminal to a folder where you want to set up the new project.

Run the commands:

```
ng new PROJECT_NAME  
cd PROJECT_NAME  
ng serve
```

That is it, you now have a simple example project made with Angular 2. You can now navigate to the link displayed in terminal and see what it is running.

To add to an existing project

Navigate to the root of your current project.

Run the command:

```
ng init
```

This will add the necessary scaffolding to your project. The files will be created in the current directory so be sure to run this in an empty directory.

Running The Project Locally

In order to see and interact with your application while it's running in the browser you must start a local development server hosting the files for your project.

```
ng serve
```

If the server started successfully it should display an address at which the server is running. Usually is this:

```
http://localhost:4200
```

Out of the box this local development server is hooked up with Hot Module Reloading, so any changes to the html, typescript, or css, will trigger the browser to be automatically reloaded (but can be disabled if desired).

Generating Components, Directives, Pipes and Services

The `ng generate <scaffold-type> <name>` (or simply `ng g <scaffold-type> <name>`) command allows you to automatically generate Angular components:

```
# The command below will generate a component in the folder you are currently at
ng generate component my-generated-component
# Using the alias (same outcome as above)
ng g component my-generated-component
```

There are several possible types of scaffolds angular-cli can generate:

Scaffold Type	Usage
Module	<code>ng g module my-new-module</code>

Scaffold Type	Usage
Component	<code>ng g component my-new-component</code>
Directive	<code>ng g directive my-new-directive</code>
Pipe	<code>ng g pipe my-new-pipe</code>
Service	<code>ng g service my-new-service</code>
Class	<code>ng g class my-new-class</code>
Interface	<code>ng g interface my-new-interface</code>
Enum	<code>ng g enum my-new-enum</code>

You can also replace the type name by its first letter. For example:

`ng g m my-new-module` to generate a new module or `ng g c my-new-component` to create a component.

Building/Bundling

When you are all finished building your Angular 2 web app and you would like to install it on a web server like Apache Tomcat, all you need to do is run the build command either with or without the production flag set. Production will minify the code and optimize for a production setting.

```
ng build
```

or

```
ng build --prod
```

Then look in the projects root directory for a `/dist` folder, which contains the build.

If you'd like the benefits of a smaller production bundle, you can also use Ahead-of-Time template compilation, which removes the template compiler from the final build:

```
ng build --prod --aot
```

Unit Testing

Angular 2 provides built-in unit testing, and every item created by angular-cli generates a basic unit test, that can be expanded. The unit tests are written using jasmine, and executed through Karma. In order to start testing execute the following command:

```
ng test
```

This command will execute all the tests in the project, and will re-execute them every time a source file changes, whether it is a test or code from the application.

For more info also visit: [angular-cli github page](#)

Getting started with Angular 2 without angular-cli.

Angular 2.0.0-rc.4

In this example we'll create a "Hello World!" app with only one root component (`AppComponent`) for the sake of simplicity.

Prerequisites:

- [Node.js](#) v5 or later
- npm v3 or later

Note: You can check versions by running `node -v` and `npm -v` in the console/terminal.

Step 1

Create and enter a new folder for your project. Let's call it `angular2-example`.

```
mkdir angular2-example
cd angular2-example
```

Step 2

Before we start writing our app code, we'll add the 4 files provided below: `package.json`, `tsconfig.json`, `typings.json`, and `systemjs.config.js`.

Disclaimer: The same files can be found in the [Official 5 Minute Quickstart](#).

`package.json` - Allows us to download all dependencies with npm and provides simple script execution to make life easier for simple projects. (You should consider using something like [Gulp](#) in the future to automate tasks).

```
{
  "name": "angular2-example",
  "version": "1.0.0",
  "scripts": {
    "start": "tsc && concurrently \"npm run tsc:w\" \"npm run lite\" ",
    "lite": "lite-server",
    "postinstall": "typings install",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "typings": "typings"
  },
  "license": "ISC",
  "dependencies": {
    "@angular/common": "2.0.0-rc.4",
    "@angular/compiler": "2.0.0-rc.4",
    "@angular/core": "2.0.0-rc.4",
    "@angular/forms": "0.2.0",
```

```

"@angular/http": "2.0.0-rc.4",
"@angular/platform-browser": "2.0.0-rc.4",
"@angular/platform-browser-dynamic": "2.0.0-rc.4",
"@angular/router": "3.0.0-beta.1",
"@angular/router-deprecated": "2.0.0-rc.2",
"@angular/upgrade": "2.0.0-rc.4",
"systemjs": "0.19.27",
"core-js": "^2.4.0",
"reflect-metadata": "^0.1.3",
"rxjs": "5.0.0-beta.6",
"zone.js": "^0.6.12",
"angular2-in-memory-web-api": "0.0.14",
"bootstrap": "^3.3.6"
},
"devDependencies": {
  "concurrently": "^2.0.0",
  "lite-server": "^2.2.0",
  "typescript": "^1.8.10",
  "typings": "^1.0.4"
}
}

```

tsconfig.json - Configures the TypeScript transpiler.

```

{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false
  }
}

```

typings.json - Makes TypeScript recognize libraries we're using.

```

{
  "globalDependencies": {
    "core-js": "registry:dt/core-js#0.0.0+20160602141332",
    "jasmine": "registry:dt/jasmine#2.2.0+20160621224255",
    "node": "registry:dt/node#6.0.0+20160621231320"
  }
}

```

systemjs.config.js - Configures [SystemJS](#) (you can also use [webpack](#)).

```

/**
 * System configuration for Angular 2 samples
 * Adjust as necessary for your application's needs.
 */
(function(global) {
  // map tells the System loader where to look for things
  var map = {
    'app': 'app', // 'dist',

```



```

    '@angular':                'node_modules/@angular',
    'angular2-in-memory-web-api': 'node_modules/angular2-in-memory-web-api',
    'rxjs':                    'node_modules/rxjs'
  };
  // packages tells the System loader how to load when no filename and/or no extension
  var packages = {
    'app':                      { main: 'main.js', defaultExtension: 'js' },
    'rxjs':                     { defaultExtension: 'js' },
    'angular2-in-memory-web-api': { main: 'index.js', defaultExtension: 'js' },
  };
  var ngPackageNames = [
    'common',
    'compiler',
    'core',
    'forms',
    'http',
    'platform-browser',
    'platform-browser-dynamic',
    'router',
    'router-deprecated',
    'upgrade',
  ];
  // Individual files (~300 requests):
  function packIndex(pkgName) {
    packages['@angular/' + pkgName] = { main: 'index.js', defaultExtension: 'js' };
  }
  // Bundled (~40 requests):
  function packUmd(pkgName) {
    packages['@angular/' + pkgName] = { main: '/bundles/' + pkgName + '.umd.js',
    defaultExtension: 'js' };
  }
  // Most environments should use UMD; some (Karma) need the individual index files
  var setPackageConfig = System.packageWithIndex ? packIndex : packUmd;
  // Add package entries for angular packages
  ngPackageNames.forEach(setPackageConfig);
  var config = {
    map: map,
    packages: packages
  };
  System.config(config);
})(this);

```

Step 3

Let's install the dependencies by typing

```
npm install
```

in the console/terminal.

Step 4

Create `index.html` inside of the `angular2-example` folder.

```

<html>
  <head>

```

```

<title>Angular2 example</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<!-- 1. Load libraries -->
<!-- Polyfill(s) for older browsers -->
<script src="node_modules/core-js/client/shim.min.js"></script>
<script src="node_modules/zone.js/dist/zone.js"></script>
<script src="node_modules/reflect-metadata/Reflect.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>
<!-- 2. Configure SystemJS -->
<script src="systemjs.config.js"></script>
<script>
  System.import('app').catch(function(err){ console.error(err); });
</script>
</head>
<!-- 3. Display the application -->
<body>
  <my-app></my-app>
</body>
</html>

```

Your application will be rendered between the `my-app` tags.

However, Angular still doesn't know *what* to render. To tell it that, we'll define `AppComponent`.

Step 5

Create a subfolder called `app` where we can define the components and [services](#) that make up our app. (In this case, it'll just contain the `AppComponent` code and `main.ts`.)

```
mkdir app
```

Step 6

Create the file `app/app.component.ts`

```

import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <ul>
      <li *ngFor="let message of messages">
        {{message}}
      </li>
    </ul>
  `
})
export class AppComponent {
  title = "Angular2 example";
  messages = [
    "Hello World!",
    "Another string",
    "Another one"
  ]
}

```

```
    ];  
  }  
}
```

What's happening? First, we're importing the `@Component` decorator which we use to give Angular the HTML tag and template for this component. Then, we're creating the class `AppComponent` with `title` and `messages` variables that we can use in the template.

Now let's look at that template:

```
<h1>{{title}}</h1>  
<ul>  
  <li *ngFor="let message of messages">  
    {{message}}  
  </li>  
</ul>
```

We're displaying the `title` variable in an `h1` tag and then making a list showing each element of the `messages` array by using the `*ngFor` directive. For each element in the array, `*ngFor` creates a `message` variable that we use within the `li` element. The result will be:

```
<h1>Angular 2 example</h1>  
<ul>  
  <li>Hello World!</li>  
  <li>Another string</li>  
  <li>Another one</li>  
</ul>
```

Step 7

Now we create a `main.ts` file, which will be the first file that Angular looks at.

Create the file `app/main.ts`.

```
import { bootstrap } from '@angular/platform-browser-dynamic';  
import { AppComponent } from './app.component';  
  
bootstrap(AppComponent);
```

We're importing the `bootstrap` function and `AppComponent` class, then using `bootstrap` to tell Angular which component to use as the root.

Step 8

It's time to fire up your first app. Type

```
npm start
```

in your console/terminal. This will run a prepared script from `package.json` that starts `lite-server`, opens your app in a browser window, and runs the TypeScript transpiler in watch mode (so `.ts`

files will be transpiled and the browser will refresh when you save changes).

What now?

Check out [the official Angular 2 guide](#) and the other topics on [StackOverflow's documentation](#).

You can also edit `AppComponent` to use external templates, styles or add/edit component variables. You should see your changes immediately after saving files.

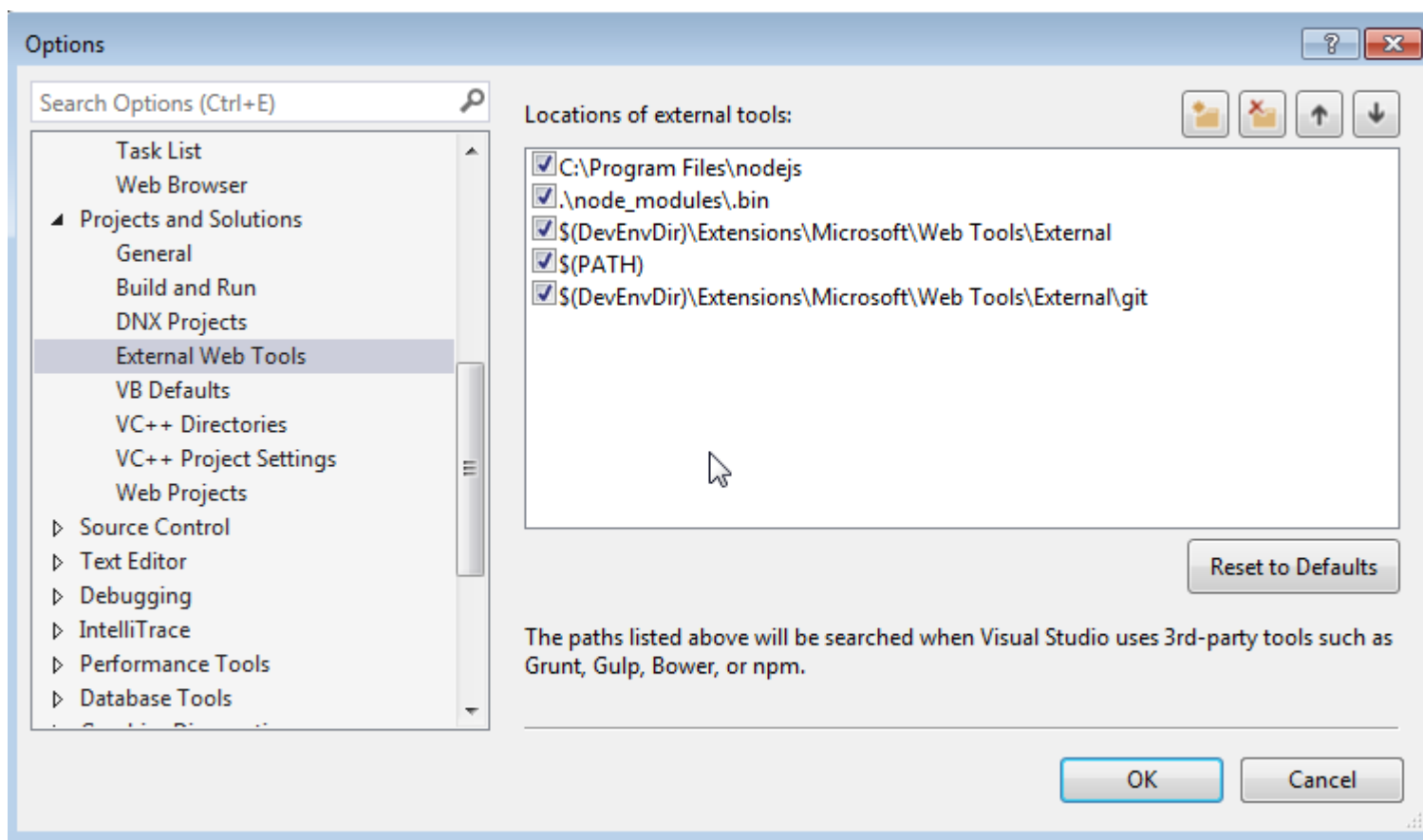
Keeping Visual Studios in sync with NPM and NODE Updates

Step 1: Locate your download of Node.js, typically it is installed under `C:/program files/nodejs`

Step 2: Open Visual Studios and navigate to "Tools>Options"

Step 3: In the options window navigate to "Projects and Solutions>External Web Tools"

Step 4: Add new entry with you Node.js file location (`C:/program files/nodejs`), IMPORTANT use the arrow buttons on menu to move your reference to the top of the list.



Step 5: Restart Visual Studios and Run an `npm install`, against your project, from `npm` command window

Getting through that pesky company proxy

If you are attempting to get an Angular2 site running on your Windows work computer at XYZ

MegaCorp the chances are that you are having problems getting through the company proxy.

There are (at least) two package managers that need to get through the proxy:

1. NPM
2. Typings

For NPM you need to add the following lines to the `.npmrc` file:

```
proxy=http://[DOMAIN]%5C[USER]:[PASS]@[PROXY]:[PROXYPORT] /
https-proxy=http://[DOMAIN]%5C[USER]:[PASS]@[PROXY]:[PROXYPORT] /
```

For Typings you need to add the following lines to the `.typingsrc` file:

```
proxy=http://[DOMAIN]%5C[USER]:[PASS]@[PROXY]:[PROXYPORT] /
https-proxy=http://[DOMAIN]%5C[USER]:[PASS]@[PROXY]:[PROXYPORT] /
rejectUnauthorized=false
```

These files probably don't exist yet, so you can create them as blank text files. They can be added to the project root (same place as `package.json` or you can put them in `%HOMEPATH%` and they will be available to all your projects.

The bit that isn't obvious and is the main reason people think the proxy settings aren't working is the `%5C` which is the URL encode of the `\` to separate the domain and user names. Thanks to Steve Roberts for that one: [Using npm behind corporate proxy .pac](#)

Getting started with Angular 2 with node.js/expressjs backend (http example included)

We will create a simple "Hello World!" app with Angular2 2.4.1 (@NgModule change) with a node.js (expressjs) backend.

Prerequisites

- [Node.js](#) v4.x.x or higher
- [npm](#) v3.x.x or higher or [yarn](#)

Then run `npm install -g typescript` or `yarn global add typescript` to install typescript globally

Roadmap

Step 1

Create a new folder (and the root dir of our back-end) for our app. Let's call it `Angular2-express`.

command line:

```
mkdir Angular2-express
cd Angular2-express
```

Step2

Create the `package.json` (for dependencies) and `app.js` (for bootstrapping) for our `node.js` app.

package.json:

```
{
  "name": "Angular2-express",
  "version": "1.0.0",
  "description": "",
  "scripts": {
    "start": "node app.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.13.3",
    "express": "^4.13.3"
  }
}
```

app.js:

```
var express = require('express');
var app = express();
var server = require('http').Server(app);
var bodyParser = require('body-parser');

server.listen(process.env.PORT || 9999, function(){
  console.log("Server connected. Listening on port: " + (process.env.PORT || 9999));
});

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));

app.use( express.static(__dirname + '/front' ) );

app.get('/test', function(req,res){ //example http request receiver
  return res.send(myTestVar);
});

//send the index.html on every page refresh and let angular handle the routing
app.get('/*', function(req, res, next) {
  console.log("Reloading");
  res.sendFile('index.html', { root: __dirname });
});
```

Then run an `npm install` or `yarn` to install the dependencies.

Now our back-end structure is complete. Let's move on to the front-end.

Step3

Our front-end should be in a folder named `front` inside our `Angular2-express` folder.

command line:

```
mkdir front
cd front
```

Just like we did with our back-end our front-end needs the dependency files too. Let's go ahead and create the following files: `package.json`, `systemjs.config.js`, `tsconfig.json`

package.json:

```
{
  "name": "Angular2-express",
  "version": "1.0.0",
  "scripts": {
    "tsc": "tsc",
    "tsc:w": "tsc -w"
  },
  "licenses": [
    {
      "type": "MIT",
      "url": "https://github.com/angular/angular.io/blob/master/LICENSE"
    }
  ],
  "dependencies": {
    "@angular/common": "~2.4.1",
    "@angular/compiler": "~2.4.1",
    "@angular/compiler-cli": "^2.4.1",
    "@angular/core": "~2.4.1",
    "@angular/forms": "~2.4.1",
    "@angular/http": "~2.4.1",
    "@angular/platform-browser": "~2.4.1",
    "@angular/platform-browser-dynamic": "~2.4.1",
    "@angular/platform-server": "^2.4.1",
    "@angular/router": "~3.4.0",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.8",
    "rxjs": "^5.0.2",
    "systemjs": "0.19.40",
    "zone.js": "^0.7.4"
  },
  "devDependencies": {
    "@types/core-js": "^0.9.34",
    "@types/node": "^6.0.45",
    "typescript": "2.0.2"
  }
}
```

systemjs.config.js:

```
/**
 * System configuration for Angular samples
 * Adjust as necessary for your application needs.
```

```

*/
(function (global) {
  System.config({
    defaultJSExtensions:true,
    paths: {
      // paths serve as alias
      'npm:': 'node_modules/'
    },
    // map tells the System loader where to look for things
    map: {
      // our app is within the app folder
      app: 'app',
      // angular bundles
      '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
      '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
      '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
      '@angular/platform-browser': 'npm:@angular/platform-browser/bundles/platform-
browser.umd.js',
      '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-
dynamic/bundles/platform-browser-dynamic.umd.js',
      '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
      '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
      '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
      // other libraries
      'rxjs':          'npm:rxjs',
      'angular-in-memory-web-api': 'npm:angular-in-memory-web-api',
    },
    // packages tells the System loader how to load when no filename and/or no extension
    packages: {
      app: {
        main: './main.js',
        defaultExtension: 'js'
      },
      rxjs: {
        defaultExtension: 'js'
      }
    }
  });
})(this);

```

tsconfig.json:

```

{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false
  },
  "compileOnSave": true,
  "exclude": [
    "node_modules/*"
  ]
}

```


Then run an `npm install` or `yarn` to install the dependencies.

Now that our dependency files are complete. Let's move on to our `index.html`:

index.html:

```
<html>
  <head>
    <base href="/">
    <title>Angular2-express</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- 1. Load libraries -->
    <!-- Polyfill(s) for older browsers -->
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <!-- 2. Configure SystemJS -->
    <script src="systemjs.config.js"></script>
    <script>
      System.import('app').catch(function(err){ console.error(err); });
    </script>

  </head>
  <!-- 3. Display the application -->
  <body>
    <my-app>Loading...</my-app>
  </body>
</html>
```

Now we're ready to create our first component. Create a folder named `app` inside our `front` folder.

command line:

```
mkdir app
cd app
```

Let's make the following files named `main.ts`, `app.module.ts`, `app.component.ts`

main.ts:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app.module';

const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

app.module.ts:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from "@angular/http";
```

```
import { AppComponent } from './app.component';

@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  declarations: [
    AppComponent
  ],
  providers: [],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

app.component.ts:

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';

@Component({
  selector: 'my-app',
  template: 'Hello World!',
  providers: []
})
export class AppComponent {
  constructor(private http: Http){
    //http get example
    this.http.get('/test')
      .subscribe((res)=>{
        console.log(res);
      });
  }
}
```

After this, compile the typescript files to javascript files. Go 2 levels up from the current dir (inside Angular2-express folder) and run the command below.

command line:

```
cd ..
cd ..
tsc -p front
```

Our folder structure should look like:

```
Angular2-express
├─ app.js
├─ node_modules
├─ package.json
├─ front
│   └─ package.json
│   └─ index.html
│   └─ node_modules
│   └─ systemjs.config.js
│   └─ tsconfig.json
│   └─ app
```

```
| | |— app.component.ts
| | |— app.component.js.map
| | |— app.component.js
| | |— app.module.ts
| | |— app.module.js.map
| | |— app.module.js
| | |— main.ts
| | |— main.js.map
| | |— main.js
```

Finally, inside Angular2-express folder, run `node app.js` command in the command line. Open your favorite browser and check `localhost:9999` to see your app.

Let's dive into Angular 4!

Angular 4 is now available! Actually Angular uses semver since Angular 2, which requires the major number being increased when breaking changes were introduced. The Angular team postponed features that cause breaking changes, which will be released with Angular 4. Angular 3 was skipped to be able to align the version numbers of the core modules, because the Router already had version 3.

As per the Angular team, Angular 4 applications are going to be less space consuming and faster than before. They have separated animation package from `@angular/core` package. If anybody is not using animation package so extra space of code will not end up in the production. The template binding syntax now supports if/else style syntax. Angular 4 is now compatible with most recent version of Typescript 2.1 and 2.2. So, Angular 4 is going to be more exciting.

Now I'll show you how to do setup of Angular 4 in your project.

Let's start Angular setup with three different ways:

You can use Angular-CLI (Command Line Interface) , It will install all dependencies for you.

- You can migrate from Angular 2 to Angular 4.
- You can use github and clone the Angular4-boilerplate. (It is the easiest one.)
- Angular Setup using Angular-CLI(command Line Interface).

Before You start using Angular-CLI , make sure You have node installed in your machine. Here, I am using node v7.8.0. Now, Open your terminal and type the following command for Angular-CLI.

```
npm install -g @angular/cli
```

or

```
yarn global add @angular/cli
```

depending on the package manager you use.

Let's install Angular 4 using Angular-CLI.

```
ng new Angular4-boilerplate
```

cd Angular4-boilerplate We are all set for Angular 4. Its pretty easy and straightforward method.

Angular Setup by migrating from Angular 2 to Angular 4

Now Let's see the second approach. I ll show you how to migrate Angular 2 to Angular 4. For that You need clone any Angular 2 project and update Angular 2 dependencies with the Angular 4 Dependency in your package.json as following:

```
"dependencies": {
  "@angular/animations": "^4.1.0",
  "@angular/common": "4.0.2",
  "@angular/compiler": "4.0.2",
  "@angular/core": "^4.0.1",
  "@angular/forms": "4.0.2",
  "@angular/http": "4.0.2",
  "@angular/material": "^2.0.0-beta.3",
  "@angular/platform-browser": "4.0.2",
  "@angular/platform-browser-dynamic": "4.0.2",
  "@angular/router": "4.0.2",
  "typescript": "2.2.2"
}
```

These are the main dependencies for Angular 4. Now You can npm install and then npm start to run the application. For reference my package.json.

Angular setup from github project

Before starting this step make sure you have git installed in your machine. Open your terminal and clone the angular4-boilerplate using below command:

```
git@github.com:CypherTree/angular4-boilerplate.git
```

Then install all dependencies and run it.

```
npm install
```

```
npm start
```

And you are done with the Angular 4 setup. All the steps are very straightforward so you can opt any of them.

Directory Structure of the angular4-boilerplate

```
Angular4-boilerplate
-karma
-node_modules
-src
  -mocks
```

```
-models
  -loginform.ts
  -index.ts
-modules
  -app
    -app.component.ts
  -app.component.html
  -login
  -login.component.ts
  -login.component.html
  -login.component.css
  -widget
  -widget.component.ts
  -widget.component.html
  -widget.component.css
  .....
-services
  -login.service.ts
  -rest.service.ts
-app.routing.module.ts
-app.module.ts
-bootstrap.ts
-index.html
-vendor.ts
-typings
-webpack
-package.json
-tsconfig.json
-tslint.json
-typings.json
```

Basic understanding for Directory structure:

All the code resides in src folder.

mocks folder is for mock data that is used in testing purpose.

model folder contains the class and interface that used in component.

modules folder contains list of components such as app, login, widget etc. All component contains typescript, html and css file. index.ts is for exporting all the class.

services folder contains list of services used in application. I have separated rest service and different component service. In rest service contains different http methods. Login service works as mediator between login component and rest service.

app.routing.ts file describes all possible routes for the application.

app.module.ts describes app module as root component.

bootstrap.ts will run the whole application.

webpack folder contains webpack configuration file.

package.json file is for all list of dependencies.

karma contains karma configuration for unit test.

node_modules contains list of package bundles.

Lets start with Login component. In login.component.html

```
<form>Dreamfactory - Addressbook 2.0
  <label>Email</label> <input id="email" form="" name="email" type="email" />
  <label>Password</label> <input id="password" form="" name="password"
  type="password" />
  <button form="">Login</button>
</form>
```

In login.component.ts

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { Form, FormGroup } from '@angular/forms';
import { LoginForm } from '../models';
import { LoginService } from '../services/login.service';

@Component({
  selector: 'login',
  template: require('./login.component.html'),
  styles: [require('./login.component.css')]
})
export class LoginComponent {

  constructor(private loginService: LoginService, private router: Router, form: LoginForm) {
  }

  getLogin(form: LoginForm): void {
    let username = form.email;
    let password = form.password;
    this.loginService.getAuthenticate(form).subscribe(() => {
      this.router.navigate(['/calender']);
    });
  }
}
```

We need to export this component to in index.ts.

```
export * from './login/login.component';
```

we need to set routes for login in app.routes.ts

```
const appRoutes: Routes = [
  {
    path: 'login',
    component: LoginComponent
  },
  .....
  {
    path: '',
    pathMatch: 'full',
    redirectTo: '/login'
  }
]
```

```
}  
];
```

In root component, app.module.ts file you just need to import that component.

```
.....  
import { LoginComponent } from './modules';  
.....  
@NgModule({  
  bootstrap: [AppComponent],  
  declarations: [  
    LoginComponent  
    .....  
    .....  
  ]  
  .....  
})  
export class AppModule { }
```

and after that npm install and npm start. Here, you go! You can check login screen in your localhost. In case of any difficulty, You can refer the angular4-boilerplate.

Basically I can feel less building package and more faster response with Angular 4 application and Although I found Exactly similar to Angular 2 in coding.

Read Getting started with Angular 2 online: <https://riptutorial.com/angular2/topic/789/getting-started-with-angular-2>

Chapter 2: Advanced Component Examples

Remarks

Remember that Angular 2 is all about singular responsibility. No matter how small your component is, dedicate a separate logic for each and every component. Be it a button, a fancy anchor link, a dialog header or even a sidenav's sub item.

Examples

Image Picker with Preview

In this example, we are going to create an image picker that previews your picture before uploading. The previewer also supports drag and dropping files into the input. In this example, I am only going to cover uploading single files, but you can tinker a bit to get multi file upload working.

image-preview.html

This is the html layout of our image preview

```
<!-- Icon as placeholder when no file picked -->
<i class="material-icons">cloud_upload</i>

<!-- file input, accepts images only. Detect when file has been picked/changed with Angular's
native (change) event listener -->
<input type="file" accept="image/*" (change)="updateSource($event)">

<!-- img placeholder when a file has been picked. shows only when 'source' is not empty -->
<img *ngIf="source" [src]="source" src="">
```

image-preview.ts

This is the main file for our `<image-preview>` component

```
import {
  Component,
  Output,
  EventEmitter,
} from '@angular/core';

@Component({
  selector: 'image-preview',
  styleUrls: [ './image-preview.css' ],
  templateUrl: './image-preview.html'
})
export class MtImagePreviewComponent {

  // Emit an event when a file has been picked. Here we return the file itself
  @Output() onChange: EventEmitter<File> = new EventEmitter<File>();
```



```

constructor() {}

// If the input has changed(file picked) we project the file into the img previewer
updateSource($event: Event) {
  // We access he file with $event.target['files'][0]
  this.projectImage($event.target['files'][0]);
}

// Uses FileReader to read the file from the input
source:string = '';
projectImage(file: File) {
  let reader = new FileReader;
  // TODO: Define type of 'e'
  reader.onload = (e: any) => {
    // Simply set e.target.result as our <img> src in the layout
    this.source = e.target.result;
    this.onChange.emit(file);
  };
  // This will process our file and get it's attributes/data
  reader.readAsDataURL(file);
}
}

```

another.component.html

```

<form (ngSubmit)="submitPhoto()">
  <image-preview (onChange)="getFile($event)"></image-preview>
  <button type="submit">Upload</button>
</form>

```

And that's it. Way more easier than it was in AngularJS 1.x. I actually made this component based on an older version I made in AngularJS 1.5.5.

Filter out table values by the input

Import `ReactiveFormsModule`, and then

```

import { Component, OnInit, OnDestroy } from '@angular/core';
import { FormControl } from '@angular/forms';
import { Subscription } from 'rxjs';

@Component({
  selector: 'component',
  template: `
    <input [formControl]="control" />
    <div *ngFor="let item of content">
      {{item.id}} - {{item.name}}
    </div>
  `
})
export class MyComponent implements OnInit, OnDestroy {

  public control = new FormControl('');

  public content: { id: number; name: string; }[];

```

```
private originalContent = [
  { id: 1, name: 'abc' },
  { id: 2, name: 'abce' },
  { id: 3, name: 'ced' }
];

private subscription: Subscription;

public ngOnInit() {
  this.subscription = this.control.valueChanges.subscribe(value => {
    this.content = this.originalContent.filter(item => item.name.startsWith(value));
  });
}

public ngOnDestroy() {
  this.subscription.unsubscribe();
}
}
```

Read Advanced Component Examples online:

<https://riptutorial.com/angular2/topic/5597/advanced-component-examples>

Chapter 3: Ahead-of-time (AOT) compilation with Angular 2

Examples

1. Install Angular 2 dependencies with compiler

NOTE: for best results, make sure your project was created using the Angular-CLI.

```
npm install angular/{core,common,compiler,platform-browser,platform-browser-dynamic,http,router,forms,compiler-cli,tsc-wrapped,platform-server}
```

You don't have to do this step if your project already has angular 2 and all of these dependencies installed. Just make sure that the `compiler` is in there.

2. Add `angularCompilerOptions` to your `tsconfig.json` file

```
...  
"angularCompilerOptions": {  
  "genDir": "./ngfactory"  
}  
...
```

This is the output folder of the compiler.

3. Run `ngc`, the angular compiler

from the root of your project `./node_modules/.bin/ngc -p src` where `src` is where all your angular 2 code lives. This will generate a folder called `ngfactory` where all your compiled code will live.

`"node_modules/.bin/ngc" -p src` for windows

4. Modify `main.ts` file to use `NgFactory` and static platform browser

```
// this is the static platform browser, the usual counterpart is @angular/platform-browser-dynamic.  
import { platformBrowser } from '@angular/platform-browser';  
  
// this is generated by the angular compiler  
import { AppModuleNgFactory } from './ngfactory/app/app.module.ngfactory';  
  
// note the use of `bootstrapModuleFactory`, as opposed to `bootstrapModule`.  
platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

At this point you should be able to run your project. In this case, my project was created using the Angular-CLI.

```
> ng serve
```

Why we need compilation, Flow of events compilation and example?

Q. Why we need compilation? Ans. We need compilation for achieving higher level of efficiency of our Angular applications.

Take a look at the following example,

```
// ...
compile: function (el, scope) {
  var dirs = this._getElDirectives(el);
  var dir;
  var scopeCreated;
  dirs.forEach(function (d) {
    dir = Provider.get(d.name + Provider.DIRECTIVES_SUFFIX);
    if (dir.scope && !scopeCreated) {
      scope = scope.$new();
      scopeCreated = true;
    }
    dir.link(el, scope, d.value);
  });
  Array.prototype.slice.call(el.children).forEach(function (c) {
    this.compile(c, scope);
  }, this);
},
// ...
```

Using the code above to render the template,

```
<ul>
  <li *ngFor="let name of names"></li>
</ul>
```

Is much slower compared to:

```
// ...
this._text_9 = this.renderer.createText(this._el_3, '\n', null);
this._text_10 = this.renderer.createText(parentRenderNode, '\n\n', null);
this._el_11 = this.renderer.createElement(parentRenderNode, 'ul', null);
this._text_12 = this.renderer.createText(this._el_11, '\n ', null);
this._anchor_13 = this.renderer.createTemplateAnchor(this._el_11, null);
this._appEl_13 = new import2.AppElement(13, 11, this, this._anchor_13);
this._TemplateRef_13_5 = new import17.TemplateRef_(this._appEl_13,
viewFactory_HomeComponent1);
this._NgFor_13_6 = new import15.NgFor(this._appEl_13.vcRef, this._TemplateRef_13_5,
this.parentInjector.get(import18.IterableDiffers), this.ref);
// ...
```

Flow of events with Ahead-of-Time Compilation

In contrast, with AoT we get through the following steps:

1. Development of Angular 2 application with TypeScript.

2. Compilation of the application with ngc.
3. Performs compilation of the templates with the Angular compiler to TypeScript.
4. Compilation of the TypeScript code to JavaScript.
5. Bundling.
6. Minification.
7. Deployment.

Although the above process seems lightly more complicated the user goes only through the steps:

1. Download all the assets.
2. Angular bootstraps.
3. The application gets rendered.

As you can see the third step is missing which means faster/better UX and on top of that tools like angular2-seed and angular-cli will automate the build process dramatically.

I hope it might help you! Thank you!

Using AoT Compilation with Angular CLI

The [Angular CLI](#) command-line interface has AoT compilation support since beta 17.

To build your app with AoT compilation, simply run:

```
ng build --prod --aot
```

Read Ahead-of-time (AOT) compilation with Angular 2 online:

<https://riptutorial.com/angular2/topic/6634/ahead-of-time--aot--compilation-with-angular-2>

Chapter 4: Angular 2 - Protractor

Examples

Testing Navbar routing with Protractor

First lets create basic navbar.html with 3 options. (Home, List , Create)

```
<nav class="navbar navbar-default" role="navigation">
<ul class="nav navbar-nav">
  <li>
    <a id="home-navbar" routerLink="/home">Home</a>
  </li>
  <li>
    <a id="list-navbar" routerLink="/create" >List</a>
  </li>
  <li>
    <a id="create-navbar" routerLink="/create">Create</a>
  </li>
</ul>
```

second lets create navbar.e2e-spec.ts

```
describe('Navbar', () => {

  beforeEach(() => {
    browser.get('home'); // before each test navigate to home page.
  });

  it('testing Navbar', () => {
    browser.sleep(2000).then(function() {
      checkNavbarTexts();
      navigateToListPage();
    });
  });

  function checkNavbarTexts() {
    element(by.id('home-navbar')).getText().then(function(text) { // Promise
      expect(text).toEqual('Home');
    });

    element(by.id('list-navbar')).getText().then(function(text) { // Promise
      expect(text).toEqual('List');
    });

    element(by.id('create-navbar')).getText().then(function(text) { // Promise
      expect(text).toEqual('Create');
    });
  }

  function navigateToListPage() {
    element(by.id('list-home')).click().then(function() { // first find list-home a tag and
    than click
      browser.sleep(2000).then(function() {
        browser.getCurrentUrl().then(function(actualUrl) { // promise
```

```

        expect(actualUrl.indexOf('list') !== -1).toBeTruthy(); // check the current url is
list
        });
    });

});
}
});

```

Angular2 Protractor - Installation

run the follows commands at cmd

- `npm install -g protractor`
- `webdriver-manager update`
- `webdriver-manager start`

****create protractor.conf.js file in the main app root.**

very important to decliare useAllAngular2AppRoots: true

```

const config = {
  baseUrl: 'http://localhost:3000/',

  specs: [
    './dev/**/*.e2e-spec.js'
  ],

  exclude: [],
  framework: 'jasmine',

  jasmineNodeOpts: {
    showColors: true,
    isVerbose: false,
    includeStackTrace: false
  },

  directConnect: true,

  capabilities: {
    browserName: 'chrome',
    shardTestFiles: false,
    chromeOptions: {
      'args': ['--disable-web-security ', '--no-sandbox', 'disable-extensions', 'start-
maximized', 'enable-crash-reporter-for-testing']
    }
  },

  onPrepare: function() {
    const SpecReporter = require('jasmine-spec-reporter');
    // add jasmine spec reporter
    jasmine.getEnv().addReporter(new SpecReporter({ displayStackTrace: true }));

    browser.ignoreSynchronization = false;
  },
  useAllAngular2AppRoots: true
};

```

```
if (process.env.TRAVIS) {
  config.capabilities = {
    browserName: 'firefox'
  };
}

exports.config = config;
```

create basic test at dev directory.

```
describe('basic test', () => {

  beforeEach(() => {
    browser.get('http://google.com');
  });

  it('testing basic test', () => {
    browser.sleep(2000).then(function() {
      browser.getCurrentUrl().then(function(actualUrl) {
        expect(actualUrl.indexOf('google') !== -1).toBeTruthy();
      });
    });
  });
});
```

run in cmd

```
protractor conf.js
```

Read Angular 2 - Protractor online: <https://riptutorial.com/angular2/topic/8900/angular--2---protractor>

Chapter 5: Angular - ForLoop

Syntax

1. `<div *ngFor="let item of items; let i = index">{{i}} {{item}}</div>`

Remarks

The `*ngFor` structural directive runs as a loop in a collection and repeats a piece of html for each element of a collection.

`@View` decorator is now deprecated. Developers should be using `template` or `'templateUrl'` properties for `@Component` decorator.

Examples

Angular 2 for-loop

For live [plnkr click...](#)

```
<!doctype html>
<html>
<head>
  <title>ng for loop in angular 2 with ES5.</title>
  <script type="text/javascript" src="https://code.angularjs.org/2.0.0-alpha.28/angular2.sfx.dev.js"></script>
  <script>
    var ngForLoop = function () {
      this.msg = "ng for loop in angular 2 with ES5.";
      this.users = ["Anil Singh", "Sunil Singh", "Sushil Singh", "Aradhya", 'Reena'];
    };

    ngForLoop.annotations = [
      new angular.Component({
        selector: 'ngforloop'
      }),
      new angular.View({
        template: '<H1>{{msg}}</H1>' +
          '<p> User List : </p>' +
          '<ul>' +
          '<li *ng-for="let user of users">' +
          '{{user}}' +
          '</li>' +
          '</ul>',
        directives: [angular.NgFor]
      })
    ];

    document.addEventListener("DOMContentLoaded", function () {
      angular.bootstrap(ngForLoop);
    });
  </script>
```

```

</head>
<body>
  <ngforloop></ngforloop>
  <h2>
    <a href="http://www.code-sample.com/" target="_blank">For more detail...</a>
  </h2>
</body>
</html>

```

NgFor - Markup For Loop

The **NgFor** directive instantiates a template once per item from an iterable. The context for each instantiated template inherits from the outer context with the given loop variable set to the current item from the iterable.

To customize the default tracking algorithm, NgFor supports **trackBy** option. **trackBy** takes a function which has two arguments: index and item. If **trackBy** is given, Angular tracks changes by the return value of the function.

```

<li *ngFor="let item of items; let i = index; trackBy: trackByFn">
  {{i}} - {{item.name}}
</li>

```

Additional Options: NgFor provides several exported values that can be aliased to local variables:

- **index** will be set to the current loop iteration for each template context.
- **first** will be set to a boolean value indicating whether the item is the first one in the iteration.
- **last** will be set to a boolean value indicating whether the item is the last one in the iteration.
- **even** will be set to a boolean value indicating whether this item has an even index.
- **odd** will be set to a boolean value indicating whether this item has an odd index.

*ngFor in the Table Rows

```

<table>
  <thead>
    <th>Name</th>
    <th>Index</th>
  </thead>
  <tbody>
    <tr *ngFor="let hero of heroes">
      <td>{{hero.name}}</td>
    </tr>
  </tbody>
</table>

```

*ngFor with component

```

@Component({
  selector: 'main-component',
  template: '<example-component

```

```

        *ngFor="let hero of heroes"
        [hero]="hero"></example-component>'
    })

@Component({
  selector: 'example-component',
  template: '<div>{{hero?.name}}</div>'
})

export class ExampleComponent {
  @Input() hero : Hero = null;
}

```

*ngFor X amount of items per row

Example shows 5 items per row:

```

<div *ngFor="let item of items; let i = index">
  <div *ngIf="i % 5 == 0" class="row">
    {{ item }}
    <div *ngIf="i + 1 < items.length">{{ items[i + 1] }}</div>
    <div *ngIf="i + 2 < items.length">{{ items[i + 2] }}</div>
    <div *ngIf="i + 3 < items.length">{{ items[i + 3] }}</div>
    <div *ngIf="i + 4 < items.length">{{ items[i + 4] }}</div>
  </div>
</div>

```

Read Angular - ForLoop online: <https://riptutorial.com/angular2/topic/6543/angular---forloop>

Chapter 6: Angular 2 Change detection and manual triggering

Examples

Basic example

Parent component :

```
import {Component} from '@angular/core';

@Component({
  selector: 'parent-component',
  templateUrl: './parent-component.html'
})
export class ParentComponent {
  users : Array<User> = [];
  changeUsersActivation(user : User){
    user.changeButtonState();
  }
  constructor(){
    this.users.push(new User('Narco', false));
    this.users.push(new User('Bombasto', false));
    this.users.push(new User('Celeritas', false));
    this.users.push(new User('Magneta', false));
  }
}

export class User {
  firstName : string;
  active : boolean;

  changeButtonState(){
    this.active = !this.active;
  }
  constructor(_firstName :string, _active : boolean){
    this.firstName = _firstName;
    this.active = _active;
  }
}
```

Parent HTML:

```
<div>
  <child-component [usersDetails]="users"
    (changeUsersActivation)="changeUsersActivation($event)">
  </child-component>
</div>
```

child component :

```

import {Component, Input, EventEmitter, Output} from '@angular/core';
import {User} from "../parent.component";

@Component({
  selector: 'child-component',
  templateUrl: './child-component.html',
  styles: [`
    .btn {
      height: 30px;
      width: 100px;
      border: 1px solid rgba(0, 0, 0, 0.33);
      border-radius: 3px;
      margin-bottom: 5px;
    }
  `]
})
export class ChildComponent {
  @Input() usersDetails : Array<User> = null;
  @Output() changeUsersActivation = new EventEmitter();

  triggerEvent(user : User) {
    this.changeUsersActivation.emit(user);
  }
}

```

child HTML :

```

<div>
  <div>
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th></th>
        </tr>
      </thead>
      <tbody *ngIf="user !== null">
        <tr *ngFor="let user of usersDetails">
          <td>{{user.firstName}}</td>
          <td><button class="btn" (click)="triggerEvent(user)">{{user.active}}</button></td>
        </tr>
      </tbody>
    </table>
  </div>
</div>

```

Read Angular 2 Change detection and manual triggering online:

<https://riptutorial.com/angular2/topic/8971/angular-2-change-detection-and-manual-triggering>

Chapter 7: Angular 2 Data Driven Forms

Remarks

```
this.myForm = this.formBuilder.group
```

creates a form object with user's configuration and assigns it to this.myForm variable.

```
'loginCredentials': this.formBuilder.group
```

method creates a group of controls which consist of a **formControlName** eg. login and value ['', Validators.required], where the first parameter is the initial value of the form input and the second is a validator or an array of validators as in 'email': ['', [Validators.required, customValidator]], .

```
'hobbies': this.formBuilder.array
```

Creates an array of groups where the index of the group is **formGroupName** in the array and is accessed like :

```
<div *ngFor="let hobby of myForm.find('hobbies').controls; let i = index">
  <div formGroupName="{{i}}">...</div>
</div>
```

```
onAddHobby() {
  (<FormArray>this.myForm.find('hobbies')).push(new FormGroup({
    'hobby': new FormControl('', Validators.required)
  }))
}
```

this sample method adds new formGroup to the array. Currently accessing requires specifying the type of control we want to access, in this example this type is : <FormArray>

```
removeHobby(index: number) {
  (<FormArray>this.myForm.find('hobbies')).removeAt(index);
}
```

same rules as above apply to removing a specific form control from the array

Examples

Data driven form

Component

```

import {Component, OnInit} from '@angular/core';
import {
  FormGroup,
  FormControl,
  FORM_DIRECTIVES,
  REACTIVE_FORM_DIRECTIVES,
  Validators,
  FormBuilder,
  FormArray
} from "@angular/forms";
import {Control} from "@angular/common";

@Component({
  moduleId: module.id,
  selector: 'app-data-driven-form',
  templateUrl: 'data-driven-form.component.html',
  styleUrls: ['data-driven-form.component.css'],
  directives: [FORM_DIRECTIVES, REACTIVE_FORM_DIRECTIVES]
})
export class DataDrivenFormComponent implements OnInit {
  myForm: FormGroup;

  constructor(private formBuilder: FormBuilder) {}

  ngOnInit() {
    this.myForm = this.formBuilder.group({
      'loginCredentials': this.formBuilder.group({
        'login': ['', Validators.required],
        'email': ['', [Validators.required, customValidator]],
        'password': ['', Validators.required]
      }),
      'hobbies': this.formBuilder.array([
        this.formBuilder.group({
          'hobby': ['', Validators.required]
        })
      ])
    });
  }

  removeHobby(index: number) {
    (<FormArray>this.myForm.find('hobbies')).removeAt(index);
  }

  onAddHobby() {
    (<FormArray>this.myForm.find('hobbies')).push(new FormGroup({
      'hobby': new FormControl('', Validators.required)
    }));
  }

  onSubmit() {
    console.log(this.myForm.value);
  }
}

function customValidator(control: Control): {[s: string]: boolean} {
  if(!control.value.match("[a-z0-9!#$%&'*/+=?^_`{|}~-]+(?:\\.[a-z0-9!#$%&'*/+=?^_`{|}~-]+)+*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?") {
    return {error: true}
  }
}

```

HTML Markup

```
<h3>Register page</h3>
<form [formGroup]="myForm" (ngSubmit)="onSubmit()">
  <div formGroupName="loginCredentials">
    <div class="form-group">
      <div>
        <label for="login">Login</label>
        <input id="login" type="text" class="form-control" formControlName="login">
      </div>
      <div>
        <label for="email">Email</label>
        <input id="email" type="text" class="form-control" formControlName="email">
      </div>
      <div>
        <label for="password">Password</label>
        <input id="password" type="text" class="form-control" formControlName="password">
      </div>
    </div>
  </div>
  <div class="row" >
    <div formGroupName="hobbies">
      <div class="form-group">
        <label>Hobbies array:</label>
        <div *ngFor="let hobby of myForm.find('hobbies').controls; let i = index">
          <div formGroupName="{{i}}">
            <input id="hobby_{{i}}" type="text" class="form-control" formControlName="hobby">
            <button *ngIf="myForm.find('hobbies').length > 1"
(click)="removeHobby(i)">x</button>
          </div>
        </div>
        <button (click)="onAddHobby()">Add hobby</button>
      </div>
    </div>
  </div>
  <button type="submit" [disabled]="!myForm.valid">Submit</button>
</form>
```

Read Angular 2 Data Driven Forms online: <https://riptutorial.com/angular2/topic/6463/angular-2-data-driven-forms>

Chapter 8: Angular 2 Forms Update

Remarks

Angular 2 allow to access the ngForm instance by creating a local template variable. Angular 2 exposes directive instances like ngForm by specifying exportAs property of the directive metadata. Now, the advantage here is without much coding you can access the ngForm instance and use it to access submitted values or to check if all the fields are valid using properties (valid, submitted, value etc).

```
#f = ngForm (creates local template instance "f")
```

ngForm emits the event "ngSubmit" when it's submitted (Check @Output documentation for more details of event emitter)

```
(ngSubmit)= "login(f.value, f.submitted) "
```

"ngModel" creates a Form Control in combination with input "name" attribute.

```
<input type="text" [(ngModel)]="username" placeholder="enter username" required>
```

When form is submitted, f.value has the JSON object representing the submitted values.

```
{ username: 'Sachin', password: 'Welcome1' }
```

Examples

Simple Password Change Form with Multi Control Validation

The below examples use the new form API introduced in RC3.

pw-change.template.html

```
<form class="container" [formGroup]="pwChangeForm">
  <label for="current">Current Password</label>
  <input id="current" formControlName="current" type="password" required><br />

  <label for="newPW">New Password</label>
  <input id="newPW" formControlName="newPW" type="password" required><br/>
  <div *ngIf="newPW.touched && newPW.newIsNotOld">
    New password can't be the same as current password.
  </div>

  <label for="confirm">Confirm new password</label>
  <input id="confirm" formControlName="confirm" type="password" required><br />
  <div *ngIf="confirm.touched && confirm.errors.newMatchesConfirm">
    The confirmation does not match.
  </div>
</form>
```

```
</div>

<button type="submit">Submit</button>
</form>
```

pw-change.component.ts

```
import {Component} from '@angular/core'
import {REACTIVE_FORM_DIRECTIVES, FormBuilder, AbstractControl, FormGroup,
  Validators} from '@angular/forms'
import {PWChangeValidators} from './pw-validators'

@Component({
  moduleId: module.id
  selector: 'pw-change-form',
  templateUrl: './pw-change.template.html`,
  directives: [REACTIVE_FORM_DIRECTIVES]
})

export class PWChangeFormComponent {
  pwChangeForm: FormGroup;

  // Properties that store paths to FormControls makes our template less verbose
  current: AbstractControl;
  newPW: AbstractControl;
  confirm: AbstractControl;

  constructor(private fb: FormBuilder) { }
  ngOnInit() {
    this.pwChangeForm = this.fb.group({
      current: ['', Validators.required],
      newPW: ['', Validators.required],
      confirm: ['', Validators.required]
    }, {
      // Here we create validators to be used for the group as a whole
      validator: Validators.compose([
        PWChangeValidators.newIsNotOld,
        PWChangeValidators.newMatchesConfirm
      ])
    });
    this.current = this.pwChangeForm.controls['current'];
    this.newPW = this.pwChangeForm.controls['newPW'];
    this.confirm = this.pwChangeForm.controls['confirm'];
  }
}
```

pw-validators.ts

```
import {FormControl, FormGroup} from '@angular/forms'
export class PWChangeValidators {

  static OldPasswordMustBeCorrect(control: FormControl) {
    var invalid = false;
    if (control.value != PWChangeValidators.oldPW)
      return { oldPasswordMustBeCorrect: true }
    return null;
  }
}
```

```

}

// Our cross control validators are below
// NOTE: They take in type FormGroup rather than FormControl
static newIsNotOld(group: FormGroup) {
    var newPW = group.controls['newPW'];
    if(group.controls['current'].value == newPW.value)
        newPW.setErrors({ newIsNotOld: true });
    return null;
}

static newMatchesConfirm(group: FormGroup) {
    var confirm = group.controls['confirm'];
    if(group.controls['newPW'].value !== confirm.value)
        confirm.setErrors({ newMatchesConfirm: true });
    return null;
}
}
}

```

A gist including some bootstrap classes can be found [here](#).

Angular 2 : Template Driven Forms

```

import { Component } from '@angular/core';
import { Router , ROUTER_DIRECTIVES} from '@angular/router';
import { NgForm } from '@angular/forms';

@Component({
    selector: 'login',
    template: `
<h2>Login</h2>
<form #f="ngForm" (ngSubmit)="login(f.value,f.valid)" novalidate>
  <div>
    <label>Username</label>
    <input type="text" [(ngModel)]="username" placeholder="enter username" required>
  </div>
  <div>
    <label>Password</label>
    <input type="password" name="password" [(ngModel)]="password" placeholder="enter password" required>
  </div>
  <input class="btn-primary" type="submit" value="Login">
</form>`
    //For long form we can use **templateUrl** instead of template
})

export class LoginComponent {

    constructor(private router : Router) { }

    login (formValue: any, valid: boolean) {
        console.log(formValue);

        if(valid) {
            console.log(valid);
        }
    }
}

```

Angular 2 Form - Custom Email/Password Validation

For live demo [click..](#)

App index ts

```
import {bootstrap} from '@angular/platform-browser-dynamic';
import {MyForm} from './my-form.component.ts';

bootstrap(MyForm);
```

Custom validator

```
import {Control} from '@angular/common';

export class CustomValidators {
  static emailFormat(control: Control): {[key: string]: boolean} {
    let pattern:RegExp = /\S+@\S+\.\S+/;
    return pattern.test(control.value) ? null : {"emailFormat": true};
  }
}
```

Form Components ts

```
import {Component} from '@angular/core';
import {FORM_DIRECTIVES, NgForm, FormBuilder, Control, ControlGroup, Validators} from
 '@angular/common';
import {CustomValidators} from './custom-validators';

@Component({
  selector: 'my-form',
  templateUrl: 'app/my-form.component.html',
  directives: [FORM_DIRECTIVES],
  styleUrls: ['styles.css']
})
export class MyForm {
  email: Control;
  password: Control;
  group: ControlGroup;

  constructor(builder: FormBuilder) {
    this.email = new Control('',
      Validators.compose([Validators.required, CustomValidators.emailFormat])
    );

    this.password = new Control('',
      Validators.compose([Validators.required, Validators.minLength(4)])
    );

    this.group = builder.group({
      email: this.email,
      password: this.password
    });
  }

  onSubmit() {
    console.log(this.group.value);
  }
}
```

```
}  
}
```

Form Components HTML

```
<form [ngFormModel]="group" (ngSubmit)="onSubmit()" novalidate>  
  
  <div>  
    <label for="email">Email:</label>  
    <input type="email" id="email" [ngFormControl]="email">  
  
    <ul *ngIf="email.dirty && !email.valid">  
      <li *ngIf="email.hasError('required')">An email is required</li>  
    </ul>  
  </div>  
  
  <div>  
    <label for="password">Password:</label>  
    <input type="password" id="password" [ngFormControl]="password">  
  
    <ul *ngIf="password.dirty && !password.valid">  
      <li *ngIf="password.hasError('required')">A password is required</li>  
      <li *ngIf="password.hasError('minlength')">A password needs to have at least 4  
characters</li>  
    </ul>  
  </div>  
  
  <button type="submit">Register</button>  
  
</form>
```

Angular 2: Reactive Forms (a.k.a Model-driven Forms)

This example uses Angular 2.0.0 Final Release

registration-form.component.ts

```
import { FormGroup,  
  FormControl,  
  FormBuilder,  
  Validators } from '@angular/forms';  
  
@Component({  
  templateUrl: './registration-form.html'  
})  
export class ExampleComponent {  
  constructor(private _fb: FormBuilder) { }  
  
  exampleForm = this._fb.group({  
    name: ['DefaultValue', [<any>Validators.required, <any>Validators.minLength(2)]],  
    email: ['default@default', [<any>Validators.required, <any>Validators.minLength(2)]]  
  })  
}
```

registration-form.html

```
<form [formGroup]="exampleForm" novalidate (ngSubmit)="submit(exampleForm)">
  <label>Name: </label>
  <input type="text" formControlName="name"/>
  <label>Email: </label>
  <input type="email" formControlName="email"/>
  <button type="submit">Submit</button>
</form>
```

Angular 2 Forms (Reactive Forms) with registration form and confirm password validation

app.module.ts

Add these into your app.module.ts file to use reactive forms

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
  ],
  declarations: [
    AppComponent
  ]
  providers: [],
  bootstrap: [
    AppComponent
  ]
})
export class AppModule {}
```

app.component.ts

```
import { Component, OnInit } from '@angular/core';
import template from './add.component.html';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';
import { matchingPasswords } from './validators';
@Component({
  selector: 'app',
  template
})
export class AppComponent implements OnInit {
  addForm: FormGroup;
  constructor(private formBuilder: FormBuilder) {
  }
  ngOnInit() {

    this.addForm = this.formBuilder.group({
      username: ['', Validators.required],
      email: ['', Validators.required],
      role: ['', Validators.required],
```

```

        password: ['', Validators.required],
        password2: ['', Validators.required] },
    { validator: matchingPasswords('password', 'password2')
    })
  });

addUser() {
  if (this.addForm.valid) {
    var adduser = {
      username: this.addForm.controls['username'].value,
      email: this.addForm.controls['email'].value,
      password: this.addForm.controls['password'].value,
      profile: {
        role: this.addForm.controls['role'].value,
        name: this.addForm.controls['username'].value,
        email: this.addForm.controls['email'].value
      }
    };
    console.log(adduser); // adduser var contains all our form values. store it where you
    want
    this.addForm.reset(); // this will reset our form values to null
  }
}
}
}

```

app.component.html

```

<div>
  <form [formGroup]="addForm">
    <input type="text" placeholder="Enter username" formControlName="username" />
    <input type="text" placeholder="Enter Email Address" formControlName="email"/>
    <input type="password" placeholder="Enter Password" formControlName="password" />
    <input type="password" placeholder="Confirm Password" name="password2"
    formControlName="password2"/>
    <div class='error' *ngIf="addForm.controls.password2.touched">
      <div class="alert-danger errorMessageadduser"
      *ngIf="addForm.hasError('mismatchedPasswords') "> Passwords do
      not match
    </div>
  </div>
  <select name="Role" formControlName="role">
    <option value="admin" >Admin</option>
    <option value="Accounts">Accounts</option>
    <option value="guest">Guest</option>
  </select>
  <br/>
  <br/>
  <button type="submit" (click)="addUser()"><span><i class="fa fa-user-plus" aria-
  hidden="true"></i></span> Add User </button>
</form>
</div>

```

validators.ts

```

export function matchingPasswords(passwordKey: string, confirmPasswordKey: string) {
  return (group: ControlGroup): {

```

```

    [key: string]: any
  } => {
    let password = group.controls[passwordKey];
    let confirmPassword = group.controls[confirmPasswordKey];

    if (password.value !== confirmPassword.value) {
      return {
        mismatchedPasswords: true
      };
    }
  }
}

```

Angular2 - Form Builder

FormComponent.ts

```

import {Component} from "@angular/core";
import {FormBuilder} from "@angular/forms";

@Component({
  selector: 'app-form',
  templateUrl: './form.component.html',
  styleUrls: ['./form.component.scss'],
  providers : [FormBuilder]
})

export class FormComponent{
  form : FormGroup;
  emailRegex = /^@w+([\.-]?\w+)*@w+([\.-]?\w+)*(\.\w{2,3})+$/;

  constructor(fb: FormBuilder) {

    this.form = fb.group({
      FirstName : new FormControl({value: null}, Validators.compose([Validators.required,
Validators.maxLength(15)])),
      LastName : new FormControl({value: null}, Validators.compose([Validators.required,
Validators.maxLength(15)])),
      Email : new FormControl({value: null}, Validators.compose([
Validators.required,
Validators.maxLength(15),
Validators.pattern(this.emailRegex)]))
    });
  }
}

```

form.component.html

```

<form class="form-details" role="form" [formGroup]="form">
  <div class="row input-label">
    <label class="form-label" for="FirstName">First name</label>
    <input
      [formControl]="form.controls['FirstName']"
      type="text"
      class="form-control"
      id="FirstName"
      name="FirstName">
  </div>

```



```
<div class="row input-label">
  <label class="form-label" for="LastName">Last name</label>
  <input
    [formControl]="form.controls['LastName']"
    type="text"
    class="form-control"
    id="LastName"
    name="LastName">
</div>
<div class="row">
  <label class="form-label" for="Email">Email</label>
  <input
    [formControl]="form.controls['Email']"
    type="email"
    class="form-control"
    id="Email"
    name="Email">
</div>
<div class="row">
  <button
    (click)="submit()"
    role="button"
    class="btn btn-primary submit-btn"
    type="button"
    [disabled]="!form.valid">Submit</button>
</div>
</div>
</form>
```

Read Angular 2 Forms Update online: <https://riptutorial.com/angular2/topic/4607/angular-2-forms-update>

Chapter 9: Angular material design

Examples

Md2Select

Component:

```
<md2-select [(ngModel)]="item" (change)="change($event)" [disabled]="disabled">
<md2-option *ngFor="let i of items" [value]="i.value" [disabled]="i.disabled">
  {{i.name}}</md2-option>
</md2-select>
```

Select allow the user to select option from options.

```
<md2-select></md2-select>
<md2-option></md2-option>
<md2-select-header></md2-select-header>
```

Md2Tooltip

Tooltip is a directive, it allows the user to show hint text while the user mouse hover over an element.

A tooltip would have the following markup.

```
<span tooltip-direction="left" tooltip="On the Left!">Left</span>
<button tooltip="some message"
  tooltip-position="below"
  tooltip-delay="1000">Hover Me
</button>
```

Md2Toast

Toast is a service, which show notifications in the view.

Creates and show a simple toast notification.

```
import {Md2Toast} from 'md2/toast/toast';

@Component({
  selector: "...",
})

export class ... {

  ...
  constructor(private toast: Md2Toast) { }
  toastMe() {
```

```
this.toast.show('Toast message...');

--- or ---

this.toast.show('Toast message...', 1000);
}

...

}
```

Md2Datepicker

Datepicker allow the user to select date and time.

```
<md2-datepicker [(ngModel)]="date"></md2-datepicker>
```

see for more details [here](#)

Md2Accordion and Md2Collapse

Md2Collapse : Collapse is a directive, it's allow the user to toggle visibility of the section.

Examples

A collapse would have the following markup.

```
<div [collapse]="isCollapsed">
  Lorem Ipsum Content
</div>
```

Md2Accordion : Accordion it's allow the user to toggle visibility of the multiple sections.

Examples

A accordion would have the following markup.

```
<md2-accordion [multiple]="multiple">
  <md2-accordion-tab *ngFor="let tab of accordions"
    [header]="tab.title"
    [active]="tab.active"
    [disabled]="tab.disabled">
    {{tab.content}}
  </md2-accordion-tab>
  <md2-accordion-tab>
    <md2-accordion-header>Custom Header</md2-accordion-header>
    test content
  </md2-accordion-tab>
</md2-accordion>
```

Read Angular material design online: <https://riptutorial.com/angular2/topic/10005/angular-material-design>

Chapter 10: angular redux

Examples

Basic

app.module.ts

```
import {appStoreProviders} from "../app.store";
providers : [
  ...
  appStoreProviders,
  ...
]
```

app.store.ts

```
import {InjectionToken} from '@angular/core';
import {createStore, Store, compose, StoreEnhancer} from 'redux';
import {AppState, default as reducer} from "../app.reducer";

export const AppStore = new InjectionToken('App.store');

const devtools: StoreEnhancer<AppState> =
  window['devToolsExtension'] ?
  window['devToolsExtension']() : f => f;

export function createAppStore(): Store<AppState> {
  return createStore<AppState>(
    reducer,
    compose(devtools)
  );
}

export const appStoreProviders = [
  {provide: AppStore, useFactory: createAppStore}
];
```

app.reducer.ts

```
export interface AppState {
  example : string
}

const rootReducer: Reducer<AppState> = combineReducers<AppState>({
  example : string
});

export default rootReducer;
```

store.ts

```

export interface IAppState {
  example?: string;
}

export const INITIAL_STATE: IAppState = {
  example: null,
};

export function rootReducer(state: IAppState = INITIAL_STATE, action: Action): IAppState {
  switch (action.type) {
    case EXAMPLE_CHANGED:
      return Object.assign(state, state, (<UpdateAction>action));
    default:
      return state;
  }
}

```

actions.ts

```

import {Action} from "redux";
export const EXAMPLE_CHANGED = 'EXAMPLE CHANGED';

export interface UpdateAction extends Action {
  example: string;
}

```

Get current state

```

import * as Redux from 'redux';
import {Inject, Injectable} from '@angular/core';

@Injectable()
export class exampleService {
  constructor(@Inject(AppStore) private store: Redux.Store<AppState>) {}
  getExampleState(){
    console.log(this.store.getState().example);
  }
}

```

change state

```

import * as Redux from 'redux';
import {Inject, Injectable} from '@angular/core';

@Injectable()
export class exampleService {
  constructor(@Inject(AppStore) private store: Redux.Store<AppState>) {}
  setExampleState(){
    this.store.dispatch(updateExample("new value"));
  }
}

```

actions.ts

```

export interface UpdateExapleAction extends Action {

```

```
    example?: string;
  }

export const updateExample: ActionCreator<UpdateExapleAction> =
  (newVal) => ({
    type: EXAMPLE_CHANGED,
    example: newVal
  });
```

Add redux chrome tool

app.store.ts

```
import {InjectionToken} from '@angular/core';
import {createStore, Store, compose, StoreEnhancer} from 'redux';
import {AppState, default as reducer} from "../app.reducer";

export const AppStore = new InjectionToken('App.store');

const devtools: StoreEnhancer<AppState> =
  window['devToolsExtension'] ?
  window['devToolsExtension']() : f => f;

export function createAppStore(): Store<AppState> {
  return createStore<AppState>(
    reducer,
    compose(devtools)
  );
}

export const appStoreProviders = [
  {provide: AppStore, useFactory: createAppStore}
];
```

install Redux DevTools chrome extention

Read angular redux online: <https://riptutorial.com/angular2/topic/10652/angular-redux>

Chapter 11: Angular RXJS Subjects and Observables with API requests

Remarks

Making API requests with Angular 2 Http service and RxJS is very similar to working with promises in Angular 1.x.

Use the `Http` class to make requests. The `Http` class exposes the methods for issuing HTTP requests `GET`, `POST`, `PUT`, `DELETE`, `PATCH`, `HEAD` requests via corresponding methods. It also exposes a generic `request` method for issuing any kind of HTTP request.

All methods of the `Http` class return an `Observable<Response>`, to which you can apply [RxJS operations](#). You call the `.subscribe()` method and pass in a function to be called when data is returned in the Observable stream.

The Observable stream for a request contains only one value - the `Response`, and completes/settles when the HTTP request is completed successfully, or errors/faults if an error is thrown.

Note, the observables returned by the `Http` module are *cold*, which means if you subscribe to the observable multiple times, the originating request will be *executed* once for each subscription. This can happen if you want to consume the result in multiple components of your application. For GET requests this might just cause some extra requests, but this can create unexpected results if subscribe more than once to PUT or POST requests.

Examples

Basic request

The following example demonstrates a simple HTTP GET request. `http.get()` returns an `Observable` which has the method `subscribe`. This one appends the returned data to the `posts` array.

```
var posts = []

getPosts(http: Http):void {
  this.http.get(`https://jsonplaceholder.typicode.com/posts`)
    .map(response => response.json())
    .subscribe(post => posts.push(post));
}
```

Encapsulating API requests

It may be a good idea to encapsulate the HTTP handling logic in its own class. The following class exposes a method for getting Posts. It calls the `http.get()` method and calls `.map` on the returned `Observable` to convert the `Response` object to a `Post` object.

```

import {Injectable} from "@angular/core";
import {Http, Response} from "@angular/http";

@Injectable()
export class BlogApi {

  constructor(private http: Http) {
  }

  getPost(id: number): Observable<Post> {
    return this.http.get(`https://jsonplaceholder.typicode.com/posts/${id}`)
      .map((response: Response) => {
        const srcData = response.json();
        return new Post(srcData)
      });
  }
}

```

The previous example uses a `Post` class to hold the returned data, which could look as follows:

```

export class Post {
  userId: number;
  id: number;
  title: string;
  body: string;

  constructor(src: any) {
    this.userId = src && src.userId;
    this.id = src && src.id;
    this.title = src && src.title;
    this.body = src && src.body;
  }
}

```

A component now can use the `BlogApi` class to easily retrieve `Post` data without concerning itself with the workings of the `Http` class.

Wait for multiple requests

One common scenario is to wait for a number of requests to finish before continuing. This can be accomplished using the [forkJoin method](#).

In the following example, `forkJoin` is used to call two methods that return `Observables`. The callback specified in the `.subscribe` method will be called when both `Observables` complete. The parameters supplied by `.subscribe` match the order given in the call to `.forkJoin`. In this case, first `posts` then `tags`.

```

loadData() : void {
  Observable.forkJoin(
    this.blogApi.getPosts(),
    this.blogApi.getTags()
  ).subscribe((([posts, tags]: [Post[], Tag[]]) => {
    this.posts = posts;
    this.tags = tags;
  }));
}

```



```
}
```

Read Angular RXJS Subjects and Observables with API requests online:

<https://riptutorial.com/angular2/topic/3577/angular-rxjs-subjects-and-observables-with-api-requests>

Chapter 12: Angular2 Animations

Introduction

Angular's animation system lets you build animations that run with the same kind of native performance found in pure CSS animations. You can also tightly integrate your animation logic with the rest of your application code, for ease of control.

Examples

Basic Animation - Transitions an element between two states driven by a model attribute.

app.component.html

```
<div>
  <div>
    <div *ngFor="let user of users">
      <button
        class="btn"
        [@buttonState]="user.active"
        (click)="user.changeButtonState()">{{user.firstName}}</button>
    </div>
  </div>
</div>
```

app.component.ts

```
import {Component, trigger, state, transition, animate, style} from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: [`
    .btn {
      height: 30px;
      width: 100px;
      border: 1px solid rgba(0, 0, 0, 0.33);
      border-radius: 3px;
      margin-bottom: 5px;
    }
  `],
  animations: [
    trigger('buttonState', [
      state('true', style({
        background: '#04b104',
        transform: 'scale(1)'
      })),
      state('false', style({
```

```

        background: '#e40202',
        transform: 'scale(1.1)'
    )),
    transition('true => false', animate('100ms ease-in')),
    transition('false => true', animate('100ms ease-out'))
  ])
]
})
export class AppComponent {
  users : Array<User> = [];
  constructor(){
    this.users.push(new User('Narco', false));
    this.users.push(new User('Bombasto', false));
    this.users.push(new User('Celeritas', false));
    this.users.push(new User('Magneta', false));
  }
}

export class User {
  firstName : string;
  active : boolean;

  changeButtonState(){
    this.active = !this.active;
  }
  constructor(_firstName :string, _active : boolean){
    this.firstName = _firstName;
    this.active = _active;
  }
}

```

Read Angular2 Animations online: <https://riptutorial.com/angular2/topic/8970/angular2-animations>

Chapter 13: Angular2 CanActivate

Examples

Angular2 CanActivate

Implemented in a router:

```
export const MainRoutes: Route[] = [{
  path: '',
  children: [ {
    path: 'main',
    component: MainComponent ,
    canActivate : [CanActivateRoute]
  } ]
}];
```

The canActivateRoute file:

```
@Injectable()
export class CanActivateRoute implements CanActivate{
  constructor(){}
  canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    return true;
  }
}
```

Read Angular2 CanActivate online: <https://riptutorial.com/angular2/topic/8899/angular2-canactivate>

Chapter 14: Angular2 Custom Validations

Parameters

parameter	description
control	This is the control that is being validated. Typically you will want to see if control.value meets some criteria.

Examples

Custom validator examples:

Angular 2 has two kinds of custom validators. Synchronous validators as in the first example that will run directly on the client and asynchronous validators (the second example) that you can use to call a remote service to do the validation for you. In this example the validator should call the server to see if a value is unique.

```
export class CustomValidators {

  static cannotContainSpace(control: Control) {
    if (control.value.indexOf(' ') >= 0)
      return { cannotContainSpace: true };

    return null;
  }

  static shouldBeUnique(control: Control) {
    return new Promise((resolve, reject) => {
      // Fake a remote validator.
      setTimeout(function () {
        if (control.value == "exisitingUser")
          resolve({ shouldBeUnique: true });
        else
          resolve(null);
      }, 1000);
    });
  }
}
```

If your control value is valid you simply return null to the caller. Otherwise you can return an object which describes the error.

Using validators in the FormBuilder

```
constructor(fb: FormBuilder) {
  this.form = fb.group({
    firstInput: ['', Validators.compose([Validators.required,
    CustomValidators.cannotContainSpace]), CustomValidators.shouldBeUnique],
    secondInput: ['', Validators.required]
  });
}
```

```
});  
}
```

Here we use the FormBuilder to create a very basic form with two input boxes. The FormBuilder takes an array for three arguments for each input control.

1. The default value of the control.
2. The validators that will run on the client. You can use `Validators.compose([arrayOfValidators])` to apply multiple validators on your control.
3. One or more async validators in a similar fashion as the second argument.

get/set FormBuilder controls parameters

There are 2 ways to set FormBuilder controls parameters.

1. On initialize:

```
exampleForm : FormGroup;  
constructor(fb: FormBuilder){  
  this.exampleForm = fb.group({  
    name : new FormControl({value: 'default name'}, Validators.compose([Validators.required,  
Validators.maxLength(15)]))  
  });  
}
```

2. After initialize:

```
this.exampleForm.controls['name'].setValue('default name');
```

Get FormBuilder control value:

```
let name = this.exampleForm.controls['name'].value();
```

Read [Angular2 Custom Validations](https://riptutorial.com/angular2/topic/6284/angular2-custom-validations) online: <https://riptutorial.com/angular2/topic/6284/angular2-custom-validations>

Chapter 15: Angular2 Databinding

Examples

@Input()

Parent Component : Initialize users lists.

```
@Component ({
  selector: 'parent-component',
  template: '<div>
    <child-component [users]="users"></child-component>
  </div>'
})
export class ParentComponent implements OnInit{
  let users : List<User> = null;

  ngOnInit() {
    users.push(new User('A', 'A', 'A@gmail.com');
    users.push(new User('B', 'B', 'B@gmail.com');
    users.push(new User('C', 'C', 'C@gmail.com');
  }
}
```

Child component get user from parent component with Input()

```
@Component ({
  selector: 'child-component',
  template: '<div>
    <table *ngIf="users !== null">
      <thead>
        <th>Name</th>
        <th>FName</th>
        <th>Email</th>
      </thead>
      <tbody>
        <tr *ngFor="let user of users">
          <td>{{user.name}}</td>
          <td>{{user.fname}}</td>
          <td>{{user.email}}</td>
        </tr>
      </tbody>
    </table>

  </div>',
})
export class ChildComponent {
  @Input() users : List<User> = null;
}

export class User {
  name : string;
```

```
fname : string;
email : string;

constructor(_name : string, _fname : string, _email : string){
  this.name = _name;
  this.fname = _fname;
  this.email = _email;
}
}
```

Read Angular2 Databinding online: <https://riptutorial.com/angular2/topic/9036/angular2-databinding>

Chapter 16: Angular2 In Memory Web API

Remarks

I mainly requested this topic because I could not find any information on setting up multiple API routes with the Angular2-In-Memory-Web-API. Ended up figuring it out myself, and figured this might be helpful to others.

Examples

Basic Setup

mock-data.ts

Create the mock api data

```
export class MockData {
  createDb() {
    let mock = [
      { id: '1', name: 'Object A' },
      { id: '2', name: 'Object B' },
      { id: '3', name: 'Object C' },
      { id: '4', name: 'Object D' }
    ];

    return {mock};
  }
}
```

main.ts

Have the dependency injector provide the InMemoryBackendService for XHRBackend requests. Also, provide a class that includes a

```
createDb()
```

function (in this case, MockData) specifying the mocked API routes for SEED_DATA requests.

```
import { XHRBackend, HTTP_PROVIDERS } from '@angular/http';
import { InMemoryBackendService, SEED_DATA } from 'angular2-in-memory-web-api';
import { MockData } from './mock-data';
import { bootstrap } from '@angular/platform-browser-dynamic';

import { AppComponent } from './app.component';

bootstrap(AppComponent, [
  HTTP_PROVIDERS,
  { provide: XHRBackend, useClass: InMemoryBackendService },
  { provide: SEED_DATA, useClass: MockData }
]);
```

mock.service.ts

Example of calling a get request for the created API route

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Mock } from './mock';

@Injectable()
export class MockService {
  // URL to web api
  private mockUrl = 'app/mock';

  constructor (private http: Http) {}

  getData(): Promise<Mock[]> {
    return this.http.get(this.mockUrl)
      .toPromise()
      .then(this.extractData)
      .catch(this.handleError);
  }

  private extractData(res: Response) {
    let body = res.json();
    return body.data || { };
  }

  private handleError (error: any) {
    let errMsg = (error.message) ? error.message :
      error.status ? `${error.status} - ${error.statusText}` : 'Server error';
    console.error(errMsg);
    return Promise.reject(errMsg);
  }
}
```

Setting Up Multiple Test API Routes

mock-data.ts

```
export class MockData {
  createDb() {
    let mock = [
      { id: '1', name: 'Object A' },
      { id: '2', name: 'Object B' },
      { id: '3', name: 'Object C' }
    ];

    let data = [
      { id: '1', name: 'Data A' },
      { id: '2', name: 'Data B' },
      { id: '3', name: 'Data C' }
    ];

    return { mock, data };
  }
}
```

Now, you can interact with both

```
app/mock
```

and

```
app/data
```

to extract their corresponding data.

Read Angular2 In Memory Web API online: <https://riptutorial.com/angular2/topic/6576/angular2-in-memory-web-api>

Chapter 17: Angular2 Input() output()

Examples

Input()

Parent Component : Initialize users lists.

```
@Component({
  selector: 'parent-component',
  template: '<div>
    <child-component [users]="users"></child-component>
  </div>'
})
export class ParentComponent implements OnInit{
  let users : List<User> = null;

  ngOnInit() {
    users.push(new User('A', 'A', 'A@gmail.com'));
    users.push(new User('B', 'B', 'B@gmail.com'));
    users.push(new User('C', 'C', 'C@gmail.com'));
  }
}
```

Child component get user from parent component with Input()

```
@Component({
  selector: 'child-component',
  template: '<div>
    <table *ngIf="users !== null">
      <thead>
        <th>Name</th>
        <th>FName</th>
        <th>Email</th>
      </thead>
      <tbody>
        <tr *ngFor="let user of users">
          <td>{{user.name}}</td>
          <td>{{user.fname}}</td>
          <td>{{user.email}}</td>
        </tr>
      </tbody>
    </table>

  </div>',
})
export class ChildComponent {
  @Input() users : List<User> = null;
}

export class User {
  name : string;
```

```
fname : string;
email : string;

constructor(_name : string, _fname : string, _email : string){
  this.name = _name;
  this.fname = _fname;
  this.email = _email;
}
}
```

Simple example of Input Properties

Parent element html

```
<child-component [isSelected]="inputPropValue"></child-component>
```

Parent element ts

```
export class AppComponent {
  inputPropValue: true
}
```

Child component ts:

```
export class ChildComponent {
  @Input() inputPropValue = false;
}
```

Child component html:

```
<div [class.simpleCssClass]="inputPropValue"></div>
```

This code will send the `inputPropValue` from the parent component to the child and it will have the value we have set in the parent component when it arrives there - false in our case. We can then use that value in the child component to, for example add a class to an element.

Read [Angular2 Input\(\) output\(\) online](https://riptutorial.com/angular2/topic/8943/angular2-input-output): <https://riptutorial.com/angular2/topic/8943/angular2-input-output>

Chapter 18: Angular2 provide external data to App before bootstrap

Introduction

In this post I will demonstrate how to pass external data to Angular app before the app bootstraps. This external data could be configuration data, legacy data, server rendered etc.

Examples

Via Dependency Injection

Instead of invoking the Angular's bootstrap code directly, wrap the bootstrap code into a function and export the function. This function can also accept parameters.

```
import { platformBrowserDynamic } from "@angular/platform-browser-dynamic";
import { AppModule } from "./src/app";
export function runAngular2App(legacyModel: any) {
  platformBrowserDynamic([
    { provide: "legacyModel", useValue: model }
  ]).bootstrapModule(AppModule)
  .then(success => console.log("Ng2 Bootstrap success"))
  .catch(err => console.error(err));
}
```

Then, in any services or components we can inject the "legacy model" and gain access to it.

```
import { Injectable } from "@angular/core";
@Injectable()
export class MyService {
  constructor(@Inject("legacyModel") private legacyModel) {
    console.log("Legacy data - ", legacyModel);
  }
}
```

Require the app and then run it.

```
require(["myAngular2App"], function(app) {
  app.runAngular2App(legacyModel); // Input to your APP
});
```

Read Angular2 provide external data to App before bootstrap online:

<https://riptutorial.com/angular2/topic/9203/angular2-provide-external-data-to-app-before-bootstrap>

Chapter 19: Angular2 using webpack

Examples

Angular 2 webpack setup

webpack.config.js

```
const webpack = require("webpack")
const helpers = require('./helpers')
const path = require("path")
const WebpackNotifierPlugin = require('webpack-notifier');

module.exports = {

  // set entry point for your app module
  "entry": {
    "app": helpers.root("app/main.module.ts"),
  },

  // output files to dist folder
  "output": {
    "filename": "[name].js",
    "path": helpers.root("dist"),
    "publicPath": "/",
  },

  "resolve": {
    "extensions": ['.ts', '.js'],
  },

  "module": {
    "rules": [
      {
        "test": /\.ts$/,
        "loaders": [
          {
            "loader": 'awesome-typescript-loader',
            "options": {
              "configFileName": helpers.root("./tsconfig.json")
            }
          },
          "angular2-template-loader"
        ]
      },
    ],
  },

  "plugins": [

    // notify when build is complete
    new WebpackNotifierPlugin({title: "build complete"}),

    // get reference for shims
    new webpack.DllReferencePlugin({
      "context": helpers.root("src/app"),
    })
  ]
}
```

```

    "manifest": helpers.root("config/polyfills-manifest.json")
  }},
  // get reference of vendor DLL
  new webpack.DllReferencePlugin({
    "context": helpers.root("src/app"),
    "manifest": helpers.root("config/vendor-manifest.json")
  }),
  // minify compiled js
  new webpack.optimize.UglifyJsPlugin(),
],
}

```

vendor.config.js

```

const webpack = require("webpack")
const helpers = require('./helpers')
const path = require("path")

module.exports = {
  // specify vendor file where all vendors are imported
  "entry": {
    // optionally add your shims as well
    "polyfills": [helpers.root("src/app/shims.ts")],
    "vendor": [helpers.root("src/app/vendor.ts")],
  },

  // output vendor to dist
  "output": {
    "filename": "[name].js",
    "path": helpers.root("dist"),
    "publicPath": "/",
    "library": "[name]"
  },

  "resolve": {
    "extensions": ['.ts', '.js'],
  },

  "module": {
    "rules": [
      {
        "test": /\.ts$/,
        "loaders": [
          {
            "loader": 'awesome-typescript-loader',
            "options": {
              "configFileName": helpers.root("./tsconfig.json")
            }
          }
        ],
      }
    ],
  },

  "plugins": [

    // create DLL for entries
    new webpack.DllPlugin({

```



```

        "name": "[name]",
        "context": helpers.root("src/app"),
        "path": helpers.root("config/[name]-manifest.json")
    }),

    // minify generated js
    new webpack.optimize.UglifyJsPlugin(),
  ],
}

```

helpers.js

```

var path = require('path');

var _root = path.resolve(__dirname, '..');

function root(args) {
  args = Array.prototype.slice.call(arguments, 0);
  return path.join.apply(path, [_root].concat(args));
}

exports.root = root;

```

vendor.ts

```

import "@angular/platform-browser"
import "@angular/platform-browser-dynamic"
import "@angular/core"
import "@angular/common"
import "@angular/http"
import "@angular/router"
import "@angular/forms"
import "rxjs"

```

index.html

```

<!DOCTYPE html>
<html>
<head>
  <title>Angular 2 webpack</title>

  <script src="/dist/vendor.js" type="text/javascript"></script>
  <script src="/dist/app.js" type="text/javascript"></script>
</head>
<body>
  <app>loading...</app>
</body>
</html>

```

package.json

```

{
  "name": "webpack example",
  "version": "0.0.0",
  "description": "webpack",
  "scripts": {

```

```
"build:webpack": "webpack --config config/webpack.config.js",
"build:vendor": "webpack --config config/vendor.config.js",
"watch": "webpack --config config/webpack.config.js --watch"
},
"devDependencies": {
  "@angular/common": "2.4.7",
  "@angular/compiler": "2.4.7",
  "@angular/core": "2.4.7",
  "@angular/forms": "2.4.7",
  "@angular/http": "2.4.7",
  "@angular/platform-browser": "2.4.7",
  "@angular/platform-browser-dynamic": "2.4.7",
  "@angular/router": "3.4.7",
  "webpack": "^2.2.1",
  "awesome-typescript-loader": "^3.1.2",
},
"dependencies": {
}
}
```

Read Angular2 using webpack online: <https://riptutorial.com/angular2/topic/9554/angular2-using-webpack>

Chapter 20: Angular-cli

Introduction

Here you will find how to start with angular-cli , generating new component/service/pipe/module with angular-cli , add 3 party like bootstrap , build angular project.

Examples

Create empty Angular2 application with angular-cli

Requirements:

- NodeJS : [Download page](#)
 - [npm](#) or [yarn](#)
-

Run the following commands with cmd from new directory folder:

1. `npm install -g @angular/cli` **OR** `yarn global add @angular/cli`
 2. `ng new PROJECT_NAME`
 3. `cd PROJECT_NAME`
 4. `ng serve`
-

Open your browser at localhost:4200

Generating Components, Directives, Pipes and Services

just use your cmd: You can use the ng generate (or just ng g) command to generate Angular components:

- **Component:** `ng g component my-new-component`
- **Directive:** `ng g directive my-new-directive`
- **Pipe:** `ng g pipe my-new-pipe`
- **Service:** `ng g service my-new-service`
- **Class:** `ng g class my-new-classt`
- **Interface:** `ng g interface my-new-interface`
- **Enum:** `ng g enum my-new-enum`
- **Module:** `ng g module my-module`

Adding 3rd party libs

In angular-cli.json you can change the app configuration.

If you want to add ng2-bootstrap for example:

1. `npm install ng2-bootstrap --save` Or `yarn add ng2-bootstrap`
2. In `angular-cli.json` just add the path of the bootstrap at `node-modules`.

```
"scripts": [  
  "../node_modules/jquery/dist/jquery.js",  
  "../node_modules/bootstrap/dist/js/bootstrap.js"  
]
```

build with angular-cli

In `angular-cli.json` at `outDir` key you can define your build directory;

these are equivalent

```
ng build --target=production --environment=prod  
ng build --prod --env=prod  
ng build --prod
```

and so are these

```
ng build --target=development --environment=dev  
ng build --dev --e=dev  
ng build --dev  
ng build
```

When building you can modify base tag (`<base>`) in your `index.html` with `--base-href your-url` option.

Sets base tag href to `/myUrl/` in your `index.html`

```
ng build --base-href /myUrl/  
ng build --bh /myUrl/
```

New project with scss/sass as stylesheet

The default style files generated and compiled by `@angular/cli` are **css**.

If you want to use **scss** instead, generate your project with:

```
ng new project_name --style=scss
```

If you want to use **sass**, generate your project with:

```
ng new project_name --style=sass
```

Set yarn as default package manager for @angular/cli

Yarn is an alternative for npm, the default package manager on `@angular/cli`. If you want to use yarn as package manager for `@angular/cli` follow this steps:

Requirements

- [yarn](#) (`npm install --global yarn` or see the [installation page](#))
- [@angular/cli](#) (`npm install -g @angular/cli` or `yarn global add @angular/cli`)

To set yarn as @angular/cli package manager:

```
ng set --global packageManager=yarn
```

To set back npm as @angular/cli package manager:

```
ng set --global packageManager=npm
```

Read Angular-cli online: <https://riptutorial.com/angular2/topic/8956/angular-cli>

Chapter 21: angular-cli test coverage

Introduction

test coverage is defined as a technique which determines whether our test cases are actually covering the application code and how much code is exercised when we run those test cases.

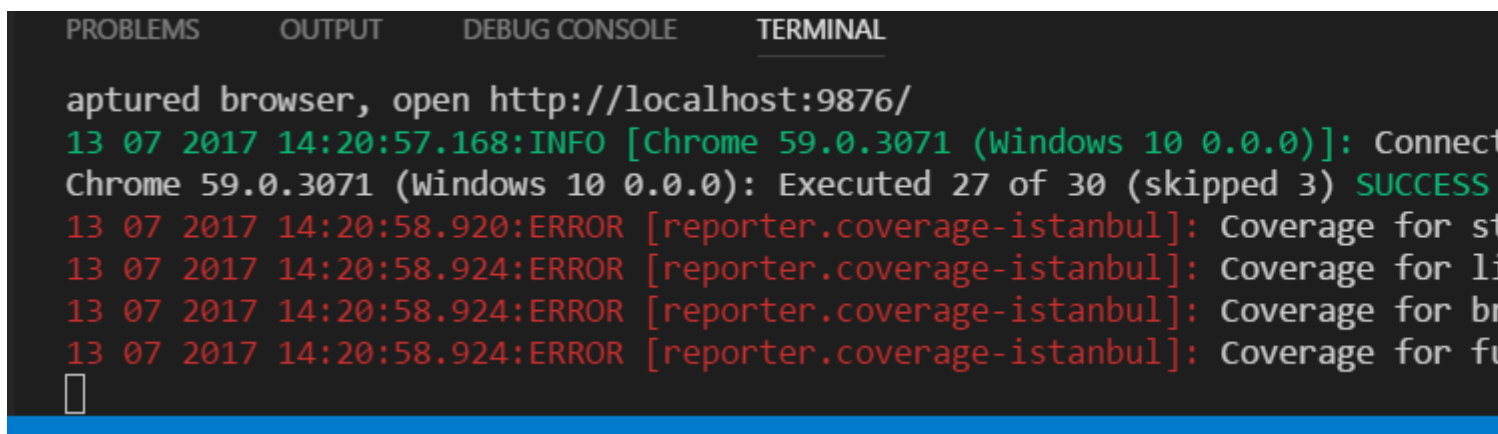
Angular CLI has built in code coverage feature with just a simple command `ng test --cc`

Examples

A simple angular-cli command base test coverage

If you want to see overall test coverage statistics than of course in Angular CLI you can just type below command, and see the bottom of your command prompt window for results.

```
ng test --cc // or --code-coverage
```



The screenshot shows an IDE interface with a terminal window. The terminal output includes the following text:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
aptured browser, open http://localhost:9876/
13 07 2017 14:20:57.168:INFO [Chrome 59.0.3071 (Windows 10 0.0.0)]: Connect
Chrome 59.0.3071 (Windows 10 0.0.0): Executed 27 of 30 (skipped 3) SUCCESS
13 07 2017 14:20:58.920:ERROR [reporter.coverage-istanbul]: Coverage for st
13 07 2017 14:20:58.924:ERROR [reporter.coverage-istanbul]: Coverage for li
13 07 2017 14:20:58.924:ERROR [reporter.coverage-istanbul]: Coverage for br
13 07 2017 14:20:58.924:ERROR [reporter.coverage-istanbul]: Coverage for fu
```

Detailed individual component base graphical test coverage reporting

if you want to see component's individual coverage of tests follow these steps.

1. `npm install --save-dev karma-teamcity-reporter`
2. Add `require('karma-teamcity-reporter')` to list of plugins in `karma.conf.js`
3. `ng test --code-coverage --reporters=teamcity,coverage-istanbul`

note that list of reporters is comma-separated, as we have added a new reporter, teamcity.

after running this command you can see the folder `coverage` in your dir and open `index.html` for a graphical view of test coverage.

All files

64.96% Statements 254/391 52.5% Branches 63/120 48.15% Functions 39/81 63.08%

File	Statements
src	
src/app/chartsapi	
src/app/echartgroup	8
src/app/echart	6
src/app/services	4
src/app	2

You can also set the coverage threshold that you want to achieve, in `karma.conf.js`, like this.

```
coverageIstanbulReporter: {
  reports: ['html', 'lcovonly'],
  fixWebpackSourcePaths: true,
  thresholds: {
    statements: 90,
    lines: 90,
    branches: 90,
    functions: 90
  }
},
```

Read angular-cli test coverage online: <https://riptutorial.com/angular2/topic/10764/angular-cli-test-coverage>

Chapter 22: Animation

Examples

Transition between null states

```
@Component({
  ...
  animations: [
    trigger('appear', [
      transition(':enter', [
        style({
          //style applied at the start of animation
        }),
        animate('300ms ease-in', style({
          //style applied at the end of animation
        })))
      ])
    ]
  })
})
class AnimComponent {
}
]
```

Animating between multiple states

The `<div>` in this template grows to 50px and then 100px and then shrinks back to 20px when you click the button.

Each `state` has an associated style described in the `@Component` metadata.

The logic for whichever `state` is active can be managed in the component logic. In this case, the component variable `size` holds the string value "small", "medium" or "large".

The `<div>` element respond to that value through the `trigger` specified in the `@Component` metadata: `[@size]="size"`.

```
@Component({
  template: '<div [@size]="size">Some Text</div><button
(click)="toggleSize()">TOGGLE</button>',
  animations: [
    trigger('size', [
      state('small', style({
        height: '20px'
      })),
      state('medium', style({
        height: '50px'
      })),
      state('large', style({
        height: '100px'
      })))
    ]
  })
})
class AnimComponent {
}
]
```



```
    )),
    transition('small => medium', animate('100ms')),
    transition('medium => large', animate('200ms')),
    transition('large => small', animate('300ms'))
  ])
]
})
export class TestComponent {

  size: string;

  constructor(){
    this.size = 'small';
  }
  toggleSize(){
    switch(this.size) {
      case 'small':
        this.size = 'medium';
        break;
      case 'medium':
        this.size = 'large';
        break;
      case 'large':
        this.size = 'small';
    }
  }
}
```

Read Animation online: <https://riptutorial.com/angular2/topic/8127/animation>

Chapter 23: Attribute directives to affect the value of properties on the host node by using the @HostBinding decorator.

Examples

@HostBinding

The @HostBinding decorator allows us to programmatically set a property value on the directive's host element. It works similarly to a property binding defined in a template, except it specifically targets the host element. The binding is checked for every change detection cycle, so it can change dynamically if desired. For example, lets say that we want to create a directive for buttons that dynamically adds a class when we press on it. That could look something like:

```
import { Directive, HostBinding, HostListener } from '@angular/core';

@Directive({
  selector: '[appButtonPress]'
})
export class ButtonPressDirective {
  @HostBinding('attr.role') role = 'button';
  @HostBinding('class.pressed') isPressed: boolean;

  @HostListener('mousedown') hasPressed() {
    this.isPressed = true;
  }
  @HostListener('mouseup') hasReleased() {
    this.isPressed = false;
  }
}
```

Notice that for both use cases of @HostBinding we are passing in a string value for which property we want to affect. If we don't supply a string to the decorator, then the name of the class member will be used instead. In the first @HostBinding, we are statically setting the role attribute to button. For the second example, the pressed class will be applied when isPressed is true

Read Attribute directives to affect the value of properties on the host node by using the @HostBinding decorator. online: <https://riptutorial.com/angular2/topic/9455/attribute-directives-to-affect-the-value-of-properties-on-the-host-node-by-using-the--hostbinding-decorator->

Chapter 24: Barrel

Introduction

A barrel is a way to rollup exports from several ES2015 modules into a single convenience ES2015 module. The barrel itself is an ES2015 module file that re-exports selected exports of other ES2015 modules.

Examples

Using Barrel

For example without a barrel, a consumer would need three import statements:

```
import { HeroComponent } from '../heroes/hero.component.ts';
import { Hero }          from '../heroes/hero.model.ts';
import { HeroService }  from '../heroes/hero.service.ts';
```

We can add a barrel by creating a file in the same component folder. In this case the folder is called 'heroes' named index.ts (using the conventions) that exports all of these items:

```
export * from './hero.model.ts'; // re-export all of its exports
export * from './hero.service.ts'; // re-export all of its exports
export { HeroComponent } from './hero.component.ts'; // re-export the named thing
```

Now a consumer can import what it needs from the barrel.

```
import { Hero, HeroService } from '../heroes/index';
```

Still, this can become a very long line; which could be reduced further.

```
import * as h from '../heroes/index';
```

That's pretty reduced! The `* as h` imports all of the modules and aliases as h

Read Barrel online: <https://riptutorial.com/angular2/topic/10717/barrel>

Chapter 25: Bootstrap Empty module in angular 2

Examples

An empty module

```
import { NgModule } from '@angular/core';

@NgModule({
  declarations: [], // components your module owns.
  imports: [], // other modules your module needs.
  providers: [], // providers available to your module.
  bootstrap: [] // bootstrap this root component.
})
export class MyModule {}
```

This is an empty module containing no declarations, imports, providers, or components to bootstrap. Use this as a reference.

A module with networking on the web browser.

```
// app.module.ts

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/http';
import { MyRootComponent } from './app.component';

@NgModule({
  declarations: [MyRootComponent],
  imports: [BrowserModule, HttpClientModule],
  bootstrap: [MyRootComponent]
})
export class MyModule {}
```

`MyRootComponent` is the root component packaged in `MyModule`. It is the entry point to your Angular 2 application.

Bootstrapping your module

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { MyModule } from './app.module';

platformBrowserDynamic().bootstrapModule( MyModule );
```

In this example, `MyModule` is the module containing your root component. By bootstrapping `MyModule` your Angular 2 app is ready to go.

Application Root Module

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }  from './app.component';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Static bootstrapping with factory classes

We can statically bootstrap an application by taking the plain ES5 Javascript output of the generated factory classes. Then we can use that output to bootstrap the application:

```
import { platformBrowser } from '@angular/platform-browser';
import { AppModuleNgFactory } from './main.ngfactory';

// Launch with the app module factory.
platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

This will cause the application bundle to be much smaller, because all the template compilation was already done in a build step, using either ngc or calling its internals directly.

Read [Bootstrap Empty module in angular 2](https://riptutorial.com/angular2/topic/5508/bootstrap-empty-module-in-angular-2) online:

<https://riptutorial.com/angular2/topic/5508/bootstrap-empty-module-in-angular-2>

Chapter 26: Brute Force Upgrading

Introduction

If you want to upgrade the Angular CLI version of your project you may run into tough-to-fix errors and bugs from simply changing the Angular CLI version number in your project. Also, because the Angular CLI hides a lot of what's going on in the build and bundles process, you can't really do much when things go wrong there.

Sometimes the easiest way to update the Angular CLI version of the project is to just scaffold out a new project with the Angular CLI version that you wish to use.

Remarks

Because Angular 2 is so modular and encapsulated you can pretty much just copy over all of your components, services, pipes, directives and then fill out the NgModule as it was in the old project.

Examples

Scaffolding a New Angular CLI Project

```
ng new NewProject
```

or

```
ng init NewProject
```

Read Brute Force Upgrading online: <https://riptutorial.com/angular2/topic/9152/brute-force-upgrading>

Chapter 27: Bypassing Sanitizing for trusted values

Parameters

Params	Details
selector	tag name you reference your component by in the html
template(templateUrl)	a string that represents html which will be inserted wherever the <code><selector></code> tag is. templateUrl is a path to an html file with the same behavior
pipes	an array of pipes that are used by this component.

Remarks

SUPER IMPORTANT!

DISABLING SANITIZING LEAVES YOU AT RISK OF XSS (Cross-Site Scripting) AND OTHER ATTACK VECTORS. PLEASE MAKE SURE YOU TRUST WHAT YOU'RE GETTING 100%

Using Pipes relegates you to only changing attribute values like so :

```
<tag [attribute]="expression or variable reference | pipeName">
```

you are not able to use pipes this way :

```
<tag attribute="expression or variable reference | pipeName">
```

or this way

```
<tag attribute={{expression or variable reference | pipeName}}>
```

Examples

Bypassing Sanitizing with pipes (for code re-use)

Project is following the structure from the Angular2 Quickstart guide [here](#).

```
RootOfProject
|
+-- app
|   |-- app.component.ts
|   |-- main.ts
|   |-- pipeUser.component.ts
|   \-- sanitize.pipe.ts
|
|-- index.html
|-- main.html
|-- pipe.html
```

main.ts

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import { AppComponent } from './app.component';

bootstrap(AppComponent);
```

This finds the index.html file in the root of the project, and builds off of that.

app.component.ts

```
import { Component } from '@angular/core';
import { PipeUserComponent } from './pipeUser.component';

@Component({
  selector: 'main-app',
  templateUrl: 'main.html',
  directives: [PipeUserComponent]
})

export class AppComponent { }
```

This is the top level component that groups other components that are used.

pipeUser.component.ts

```
import { Component } from '@angular/core';
import { IgnoreSanitize } from "./sanitize.pipe";

@Component({
  selector: 'pipe-example',
  templateUrl: "pipe.html",
  pipes: [IgnoreSanitize]
})

export class PipeUserComponent{
  constructor () { }
  unsafeValue: string = "unsafe/picUrl?id=";
  docNum: string;

  getUrl(input: string): any {
    if(input !== undefined) {
```



```

        return this.unsafeValue.concat(input);
        // returns : "unsafe/picUrl?id=input"
    } else {
        return "fallback/to/something";
    }
}
}
}

```

This component provides the view for the Pipe to work with.

sanitize.pipe.ts

```

import { Pipe, PipeTransform } from '@angular/core';
import { DomSanitizationService } from '@angular/platform-browser';

@Pipe({
  name: 'sanitaryPipe'
})
export class IgnoreSanitize implements PipeTransform {

  constructor(private sanitizer: DomSanitizationService){}

  transform(input: string) : any {
    return this.sanitizer.bypassSecurityTrustUrl(input);
  }
}

```

This is the logic that describes what the pipe formats.

index.html

```

<head>
  Stuff goes here...
</head>
<body>
  <main-app>
    main.html will load inside here.
  </main-app>
</body>

```

main.html

```

<othertags>
</othertags>

<pipe-example>
  pipe.html will load inside here.
</pipe-example>

<moretags>
</moretags>

```

pipe.html

```
<img [src]="getUrl('1234') | sanitaryPipe">
<embed [src]="getUrl() | sanitaryPipe">
```

If you were to inspect the html while the app is running you would see that it looks like this:

```
<head>
  Stuff goes here...
</head>

<body>

  <othertags>
  </othertags>

  <img [src]="getUrl('1234') | sanitaryPipe">
  <embed [src]="getUrl() | sanitaryPipe">

  <moretags>
  </moretags>

</body>
```

Read [Bypassing Sanitizing for trusted values](https://riptutorial.com/angular2/topic/5942/bypassing-sanitizing-for-trusted-values) online:

<https://riptutorial.com/angular2/topic/5942/bypassing-sanitizing-for-trusted-values>

Chapter 28: Commonly built-in directives and services

Introduction

@angular/common - commonly needed directives and services @angular/core - the angular core framework

Examples

Location Class

Location is a service that applications can use to interact with a browser's URL. Depending on which LocationStrategy is used, Location will either persist to the URL's path or the URL's hash segment.

Location is responsible for normalizing the URL against the application's base href.

```
import {Component} from '@angular/core';
import {Location} from '@angular/common';

@Component({
  selector: 'app-component'
})
class AppCmp {

  constructor(_location: Location) {

    //Changes the browsers URL to the normalized version of the given URL,
    //and pushes a new item onto the platform's history.
    _location.go('/foo');

  }

  backClicked() {
    //Navigates back in the platform's history.
    this._location.back();
  }

  forwardClicked() {
    //Navigates forward in the platform's history.
    this._location.back();
  }
}
```

AsyncPipe

The async pipe subscribes to an Observable or Promise and returns the latest value it has emitted. When a new value is emitted, the async pipe marks the component to be checked for

changes. When the component gets destroyed, the async pipe unsubscribes automatically to avoid potential memory leaks.

```
@Component({
  selector: 'async-observable-pipe',
  template: '<div><code>observable|async</code>: Time: {{ time | async }}</div>'
})
export class AsyncObservablePipeComponent {
  time = new Observable<string>((observer: Subscriber<string>) => {
    setInterval(() => observer.next(new Date().toString()), 1000);
  });
}
```

Displaying current angular2 version used in your project

To display current version, we can use **VERSION** from `@angular/core` package.

```
import { Component, VERSION } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>
<h2>Current Version: {{ver}}</h2>
`,
})
export class AppComponent {
  name = 'Angular2';
  ver = VERSION.full;
}
```

Currency Pipe

The currency pipe allows you to work with you data as regular numbers but display it with standard currency formatting (currency symbol, decimal places, etc.) in the view.

```
@Component({
  selector: 'currency-pipe',
  template: `<div>
    <p>A: {{myMoney | currency:'USD':false}}</p>
    <p>B: {{yourMoney | currency:'USD':true:'4.2-2'}}</p>
  </div>`
})
export class CurrencyPipeComponent {
  myMoney: number = 100000.653;
  yourMoney: number = 5.3495;
}
```

The pipe takes three optional parameters:

- **currencyCode**: Allows you to specify the ISO 4217 currency code.
- **symbolDisplay**: Boolean indicating whether to use the currency symbol
- **digitInfo**: Allows you to specify how the decimal places should be displayed.

More documentation on the currency pipe:

<https://angular.io/docs/ts/latest/api/common/index/CurrencyPipe-pipe.html>

Read Commonly built-in directives and services online:

<https://riptutorial.com/angular2/topic/8252/commonly-built-in-directives-and-services>

Chapter 29: Component interactions

Syntax

- `<element [variableName]="value"></element> //Declaring input to child when using @Input() method.`
- `<element (childOutput)="parentFunction($event)"></element> //Declaring output from child when using @Output() method.`
- `@Output() pageNumberClicked = new EventEmitter(); //Used for sending output data from child component when using @Output() method.`
- `this.pageNumberClicked.emit(pageNum); //Used to trigger data output from child component. when using @Output() method.`
- `@ViewChild(ComponentClass) //Property decorator is required when using ViewChild.`

Parameters

Name	Value
pageCount	Used to tell number of pages to be created to the child component.
pageNumberClicked	Name of output variable in the child component.
pageChanged	Function at parent component that listening for child components output.

Examples

Parent - Child interaction using @Input & @Output properties

We have a `DataListComponent` that shows a data we pull from a service. `DataListComponent` also has a `PagerComponent` as it's child.

`PagerComponent` creates page number list based on total number of pages it gets from the `DataListComponent`. `PagerComponent` also lets the `DataListComponent` know when user clicks any page number via `Output` property.

```
import { Component, NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { DataListService } from '../dataList.service';
import { PagerComponent } from '../pager.component';

@Component({
  selector: 'datalist',
  template: `
    <table>
    <tr *ngFor="let person of personsData">
      <td>{{person.name}}</td>
      <td>{{person.surname}}</td>
    </tr>
    </table>
```

```

        <pager [pageCount]="pageCount" (pageNumberClicked)="pageChanged($event)"></pager>
    `
})
export class DataListComponent {
    private personsData = null;
    private pageCount: number;

    constructor(private dataListService: DataListService) {
        var response = this.dataListService.getData(1); //Request first page from the service
        this.personsData = response.persons;
        this.pageCount = response.totalCount / 10; //We will show 10 records per page.
    }

    pageChanged(pageNumber: number){
        var response = this.dataListService.getData(pageNumber); //Request data from the
service with new page number
        this.personsData = response.persons;
    }
}

@NgModule({
    imports: [CommonModule],
    exports: [],
    declarations: [DataListComponent, PagerComponent],
    providers: [DataListService],
})
export class DataListModule { }

```

PagerComponent lists all the page numbers. We set click event on each of them so we can let the parent know about the clicked page number.

```

import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
    selector: 'pager',
    template: `
<div id="pager-wrapper">
    <span *ngFor="#page of pageCount" (click)="pageClicked(page)">{{page}}</span>
</div>
`
})
export class PagerComponent {
    @Input() pageCount: number;
    @Output() pageNumberClicked = new EventEmitter();
    constructor() { }

    pageClicked(pageNum) {
        this.pageNumberClicked.emit(pageNum); //Send clicked page number as output
    }
}

```

Parent - Child interaction using ViewChild

ViewChild offers one way interaction from parent to child. There is no feedback or output from child when ViewChild is used.

We have a `DataListComponent` that shows some information. `DataListComponent` has `PagerComponent` as its child. When user makes a search on `DataListComponent`, it gets a data from a service and asks `PagerComponent` to refresh paging layout based on new number of pages.

```
import { Component, NgModule, ViewChild } from '@angular/core';
import { CommonModule } from '@angular/common';
import { DataListService } from '../dataList.service';
import { PagerComponent } from '../pager.component';

@Component({
  selector: 'datalist',
  template: `<input type='text' [(ngModel)]='searchText' />
    <button (click)="getData()">Search</button>
    <table>
    <tr *ngFor="let person of personsData">
      <td>{{person.name}}</td>
      <td>{{person.surname}}</td>
    </tr>
    </table>

    <pager></pager>
  `
})
export class DataListComponent {
  private personsData = null;
  private searchText: string;

  @ViewChild(PagerComponent)
  private pagerComponent: PagerComponent;

  constructor(private dataListService: DataListService) {}

  getData() {
    var response = this.dataListService.getData(this.searchText);
    this.personsData = response.data;
    this.pagerComponent.setPaging(this.personsData / 10); //Show 10 records per page
  }
}

@NgModule({
  imports: [CommonModule],
  exports: [],
  declarations: [DataListComponent, PagerComponent],
  providers: [DataListService],
})
export class DataListModule { }
```

In this way you can call functions defined at child components.

Child component is not available until parent component is rendered. Attempting to access to the child before parents `AfterViewInit` life cycle hook will cause exception.

Bidirectional parent-child interaction through a service

Service that is used for communication:

```
import { Injectable } from '@angular/core';
```



```

import { Subject } from 'rxjs/Subject';

@Injectable()
export class ComponentCommunicationService {

  private componentChangeSource = new Subject();
  private newDateCreationSource = new Subject<Date>();

  componentChanged$ = this.componentChangeSource.asObservable();
  dateCreated$ = this.newDateCreationSource.asObservable();

  refresh() {
    this.componentChangeSource.next();
  }

  broadcastDate(date: Date) {
    this.newDateCreationSource.next(date);
  }
}

```

Parent component:

```

import { Component, Inject } from '@angular/core';
import { ComponentCommunicationService } from './component-refresh.service';

@Component({
  selector: 'parent',
  template: `
<button (click)="refreshSubscribed()">Refresh</button>
<h1>Last date from child received: {{lastDate}}</h1>
<child-component></child-component>
`
})
export class ParentComponent implements OnInit {

  lastDate: Date;
  constructor(private communicationService: ComponentCommunicationService) { }

  ngOnInit() {
    this.communicationService.dateCreated$.subscribe(newDate => {
      this.lastDate = newDate;
    });
  }

  refreshSubscribed() {
    this.communicationService.refresh();
  }
}

```

Child component:

```

import { Component, OnInit, Inject } from '@angular/core';
import { ComponentCommunicationService } from './component-refresh.service';

@Component({
  selector: 'child-component',
  template: `
<h1>Last refresh from parent: {{lastRefreshed}}</h1>
<button (click)="sendNewDate()">Send new date</button>
`
})

```

```

    })
    export class ChildComponent implements OnInit {

        lastRefreshed: Date;
        constructor(private communicationService: ComponentCommunicationService) { }

        ngOnInit() {
            this.communicationService.componentChanged$.subscribe(event => {
                this.onRefresh();
            });
        }

        sendNewDate() {
            this.communicationService.broadcastDate(new Date());
        }

        onRefresh() {
            this.lastRefreshed = new Date();
        }
    }
}

```

AppModule:

```

@NgModule({
  declarations: [
    ParentComponent,
    ChildComponent
  ],
  providers: [ComponentCommunicationService],
  bootstrap: [AppComponent] // not included in the example
})
export class AppModule {}

```

Read Component interactions online: <https://riptutorial.com/angular2/topic/7400/component-interactions>

Chapter 30: Component interactions

Introduction

Share information between different directives and components.

Examples

Pass data from parent to child with input binding

HeroChildComponent has two input properties, typically adorned with @Input decorations.

```
import { Component, Input } from '@angular/core';
import { Hero } from './hero';
@Component({
  selector: 'hero-child',
  template: `
    <h3>{{hero.name}} says:</h3>
    <p>I, {{hero.name}}, am at your service, {{masterName}}.</p>
  `
})
export class HeroChildComponent {
  @Input() hero: Hero;
  @Input('master') masterName: string;
}
```

Intercept input property changes with a setter

Use an input property setter to intercept and act upon a value from the parent.

The setter of the name input property in the child NameChildComponent trims the whitespace from a name and replaces an empty value with default text.

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'name-child',
  template: '<h3>{{name}}</h3>'
})
export class NameChildComponent {
  private _name = '';
  @Input()
  set name(name: string) {
    this._name = (name && name.trim()) || '<no name set>';
  }
  get name(): string { return this._name; }
}
```

Here's the NameParentComponent demonstrating name variations including a name with all spaces:

```
import { Component } from '@angular/core';
@Component({
  selector: 'name-parent',
  template: `
    <h2>Master controls {{names.length}} names</h2>
    <name-child *ngFor="let name of names" [name]="name"></name-child>
  `
})
export class NameParentComponent {
  // Displays 'Mr. IQ', '<no name set>', 'Bombasto'
  names = ['Mr. IQ', ' ', ' Bombasto '];
}
```

Parent listens for child event

The child component exposes an `EventEmitter` property with which it emits events when something happens. The parent binds to that event property and reacts to those events.

The child's `EventEmitter` property is an output property, typically adorned with an `@Output` decoration as seen in this `VoterComponent`:

```
import { Component, EventEmitter, Input, Output } from '@angular/core';
@Component({
  selector: 'my-voter',
  template: `
    <h4>{{name}}</h4>
    <button (click)="vote(true)" [disabled]="voted">Agree</button>
    <button (click)="vote(false)" [disabled]="voted">Disagree</button>
  `
})
export class VoterComponent {
  @Input() name: string;
  @Output() onVoted = new EventEmitter<boolean>();
  voted = false;
  vote(agreed: boolean) {
    this.onVoted.emit(agreed);
    this.voted = true;
  }
}
```

Clicking a button triggers emission of a true or false (the boolean payload).

The parent `VoteTakerComponent` binds an event handler (`onVoted`) that responds to the child event payload (`$event`) and updates a counter.

```
import { Component } from '@angular/core';
@Component({
  selector: 'vote-taker',
  template: `
    <h2>Should mankind colonize the Universe?</h2>
    <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>
    <my-voter *ngFor="let voter of voters"
      [name]="voter"
      (onVoted)="onVoted($event)">
    </my-voter>
  `
})
```

```

export class VoteTakerComponent {
  agreed = 0;
  disagreed = 0;
  voters = ['Mr. IQ', 'Ms. Universe', 'Bombasto'];
  onVoted(agreed: boolean) {
    agreed ? this.agreed++ : this.disagreed++;
  }
}

```

Parent interacts with child via local variable

A parent component cannot use data binding to read child properties or invoke child methods. We can do both by creating a template reference variable for the child element and then reference that variable within the parent template as seen in the following example.

We have a child `CountdownTimerComponent` that repeatedly counts down to zero and launches a rocket. It has `start` and `stop` methods that control the clock and it displays a countdown status message in its own template.

```

import { Component, OnDestroy, OnInit } from '@angular/core';
@Component({
  selector: 'countdown-timer',
  template: '<p>{{message}}</p>'
})
export class CountdownTimerComponent implements OnInit, OnDestroy {
  intervalId = 0;
  message = '';
  seconds = 11;
  clearTimer() { clearInterval(this.intervalId); }
  ngOnInit() { this.start(); }
  ngOnDestroy() { this.clearTimer(); }
  start() { this.countDown(); }
  stop() {
    this.clearTimer();
    this.message = `Holding at T-${this.seconds} seconds`;
  }
  private countDown() {
    this.clearTimer();
    this.intervalId = window.setInterval(() => {
      this.seconds -= 1;
      if (this.seconds === 0) {
        this.message = 'Blast off!';
      } else {
        if (this.seconds < 0) { this.seconds = 10; } // reset
        this.message = `T-${this.seconds} seconds and counting`;
      }
    }, 1000);
  }
}

```

Let's see the `CountdownLocalVarParentComponent` that hosts the timer component.

```

import { Component } from '@angular/core';
import { CountdownTimerComponent } from './countdown-timer.component';
@Component({
  selector: 'countdown-parent-lv',
  template: `

```

```

<h3>Countdown to Liftoff (via local variable)</h3>
<button (click)="timer.start()">Start</button>
<button (click)="timer.stop()">Stop</button>
<div class="seconds">{{timer.seconds}}</div>
<countdown-timer #timer></countdown-timer>
`,
styleUrls: ['demo.css']
})
export class CountdownLocalVarParentComponent { }

```

The parent component cannot data bind to the child's start and stop methods nor to its seconds property.

We can place a local variable (`#timer`) on the tag (`()`) representing the child component. That gives us a reference to the child component itself and the ability to access any of its properties or methods from within the parent template.

In this example, we wire parent buttons to the child's start and stop and use interpolation to display the child's seconds property.

Here we see the parent and child working together.

Parent calls a ViewChild

The local variable approach is simple and easy. But it is limited because the parent-child wiring must be done entirely within the parent template. The parent component itself has no access to the child.

We can't use the local variable technique if an instance of the parent component class must read or write child component values or must call child component methods.

When the parent component class requires that kind of access, we inject the child component into the parent as a `ViewChild`.

We'll illustrate this technique with the same Countdown Timer example. We won't change its appearance or behavior. The child `CountdownTimerComponent` is the same as well.

We are switching from the local variable to the `ViewChild` technique solely for the purpose of demonstration. Here is the parent, `CountdownViewChildParentComponent`:

```

import { AfterViewInit, ViewChild } from '@angular/core';
import { Component } from '@angular/core';
import { CountdownTimerComponent } from './countdown-timer.component';
@Component({
  selector: 'countdown-parent-vc',
  template: `
<h3>Countdown to Liftoff (via ViewChild)</h3>
<button (click)="start()">Start</button>
<button (click)="stop()">Stop</button>
<div class="seconds">{{ seconds() }}</div>
<countdown-timer></countdown-timer>
`,
styleUrls: ['demo.css']
})

```

```

export class CountdownViewChildParentComponent implements AfterViewInit {
  @ViewChild(CountdownTimerComponent)
  private timerComponent: CountdownTimerComponent;
  seconds() { return 0; }
  ngAfterViewInit() {
    // Redefine `seconds()` to get from the `CountdownTimerComponent.seconds` ...
    // but wait a tick first to avoid one-time devMode
    // unidirectional-data-flow-violation error
    setTimeout(() => this.seconds = () => this.timerComponent.seconds, 0);
  }
  start() { this.timerComponent.start(); }
  stop() { this.timerComponent.stop(); }
}

```

It takes a bit more work to get the child view into the parent component class.

We import references to the `ViewChild` decorator and the `AfterViewInit` lifecycle hook.

We inject the child `CountdownTimerComponent` into the private `timerComponent` property via the `@ViewChild` property decoration.

The `#timer` local variable is gone from the component metadata. Instead we bind the buttons to the parent component's own `start` and `stop` methods and present the ticking seconds in an interpolation around the parent component's `seconds` method.

These methods access the injected timer component directly.

The `ngAfterViewInit` lifecycle hook is an important wrinkle. The timer component isn't available until after Angular displays the parent view. So we display 0 seconds initially.

Then Angular calls the `ngAfterViewInit` lifecycle hook at which time it is too late to update the parent view's display of the countdown seconds. Angular's unidirectional data flow rule prevents us from updating the parent view's in the same cycle. We have to wait one turn before we can display the seconds.

We use `setTimeout` to wait one tick and then revise the `seconds` method so that it takes future values from the timer component.

Parent and children communicate via a service

A parent component and its children share a service whose interface enables bi-directional communication within the family.

The scope of the service instance is the parent component and its children. Components outside this component subtree have no access to the service or their communications.

This `MissionService` connects the `MissionControlComponent` to multiple `AstronautComponent` children.

```

import { Injectable } from '@angular/core';
import { Subject } from 'rxjs/Subject';
@Injectable()
export class MissionService {

```

```

// Observable string sources
private missionAnnouncedSource = new Subject<string>();
private missionConfirmedSource = new Subject<string>();
// Observable string streams
missionAnnounced$ = this.missionAnnouncedSource.asObservable();
missionConfirmed$ = this.missionConfirmedSource.asObservable();
// Service message commands
announceMission(mission: string) {
  this.missionAnnouncedSource.next(mission);
}
confirmMission(astronaut: string) {
  this.missionConfirmedSource.next(astronaut);
}
}

```

The `MissionControlComponent` both provides the instance of the service that it shares with its children (through the providers metadata array) and injects that instance into itself through its constructor:

```

import { Component }      from '@angular/core';
import { MissionService } from './mission.service';
@Component({
  selector: 'mission-control',
  template: `
    <h2>Mission Control</h2>
    <button (click)="announce()">Announce mission</button>
    <my-astronaut *ngFor="let astronaut of astronauts"
      [astronaut]="astronaut">
    </my-astronaut>
    <h3>History</h3>
    <ul>
      <li *ngFor="let event of history">{{event}}</li>
    </ul>
  `,
  providers: [MissionService]
})
export class MissionControlComponent {
  astronauts = ['Lovell', 'Swigert', 'Haise'];
  history: string[] = [];
  missions = ['Fly to the moon!',
    'Fly to mars!',
    'Fly to Vegas!'];
  nextMission = 0;
  constructor(private missionService: MissionService) {
    missionService.missionConfirmed$.subscribe(
      astronaut => {
        this.history.push(`${astronaut} confirmed the mission`);
      });
  }
  announce() {
    let mission = this.missions[this.nextMission++];
    this.missionService.announceMission(mission);
    this.history.push(`Mission "${mission}" announced`);
    if (this.nextMission >= this.missions.length) { this.nextMission = 0; }
  }
}

```

The `AstronautComponent` also injects the service in its constructor. Each `AstronautComponent` is

a child of the MissionControlComponent and therefore receives its parent's service instance:

```
import { Component, Input, OnDestroy } from '@angular/core';
import { MissionService } from './mission.service';
import { Subscription } from 'rxjs/Subscription';
@Component({
  selector: 'my-astronaut',
  template: `
    <p>
      {{astronaut}}: <strong>{{mission}}</strong>
      <button
        (click)="confirm()"
        [disabled]="!announced || confirmed">
        Confirm
      </button>
    </p>
  `
})
export class AstronautComponent implements OnDestroy {
  @Input() astronaut: string;
  mission = '<no mission announced>';
  confirmed = false;
  announced = false;
  subscription: Subscription;
  constructor(private missionService: MissionService) {
    this.subscription = missionService.missionAnnounced$.subscribe(
      mission => {
        this.mission = mission;
        this.announced = true;
        this.confirmed = false;
      }
    );
  }
  confirm() {
    this.confirmed = true;
    this.missionService.confirmMission(this.astronaut);
  }
  ngOnDestroy() {
    // prevent memory leak when component destroyed
    this.subscription.unsubscribe();
  }
}
```

Notice that we capture the subscription and unsubscribe when the AstronautComponent is destroyed. This is a memory-leak guard step. There is no actual risk in this app because the lifetime of a AstronautComponent is the same as the lifetime of the app itself. That would not always be true in a more complex application.

We do not add this guard to the MissionControlComponent because, as the parent, it controls the lifetime of the MissionService. The History log demonstrates that messages travel in both directions between the parent MissionControlComponent and the AstronautComponent children, facilitated by the service:

Read Component interactions online: <https://riptutorial.com/angular2/topic/9454/component-interactions>

Chapter 31: Components

Introduction

Angular components are elements composed by a template that will render your application.

Examples

A simple component

To create a component we add `@Component` decorator in a class passing some parameters:

- `providers`: Resources that will be injected into the component constructor
- `selector`: The query selector that will find the element in the HTML and replace by the component
- `styles`: Inline styles. NOTE: DO NOT use this parameter with `require`, it works on development but when you build the application in production all your styles are lost
- `styleUrls`: Array of path to style files
- `template`: String that contains your HTML
- `templateUrl`: Path to a HTML file

There are other parameters you can configure, but the listed ones are what you will use the most.

A simple example:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-required',
  styleUrls: ['required.component.scss'],
  // template: `This field is required.`,
  templateUrl: 'required.component.html',
})
export class RequiredComponent { }
```

Templates & Styles

Templates are HTML files that may contain logic.

You can specify a template in two ways:

Passing template as a file path

```
@Component({
  templateUrl: 'hero.component.html',
})
```

Passing a template as an inline code

```
@Component ({
  template: `
```

Templates may contain styles. The styles declared in `@Component` are different from your application style file, anything applied in the component will be restricted to this scope. For example, say you add:

```
div { background: red; }
```

All `div`s inside the component will be red, but if you have other components, other `div`s in your HTML they will not be changed at all.

The generated code will look like this:

```
<style div[_ngcontent-c1] { background: red; }</style>
```

You can add styles to a component in two ways:

Passing an array of file paths

```
@Component ({
  styleUrls: ['hero.component.css'],
})
```

Passing an array of inline codes

```
styles: [ `div { background: lime; }` ]
```

You shouldn't use `styles` with `require` as it will not work when you build your application to production.

Testing a Component

hero.component.html

```
<form (ngSubmit)="submit($event)" [formGroup]="form" novalidate>
  <input type="text" FormControlName="name" />
  <button type="submit">Show hero name</button>
</form>
```

hero.component.ts

```
import { FormControl, FormGroup, Validators } from '@angular/forms';
```

```

import { Component } from '@angular/core';

@Component({
  selector: 'app-hero',
  templateUrl: 'hero.component.html',
})
export class HeroComponent {
  public form = new FormGroup({
    name: new FormControl('', Validators.required),
  });

  submit(event) {
    console.log(event);
    console.log(this.form.controls.name.value);
  }
}

```

hero.component.spec.ts

```

import { ComponentFixture, TestBed, async } from '@angular/core/testing';

import { HeroComponent } from './hero.component';
import { ReactiveFormsModule } from '@angular/forms';

describe('HeroComponent', () => {
  let component: HeroComponent;
  let fixture: ComponentFixture<HeroComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [HeroComponent],
      imports: [ReactiveFormsModule],
    }).compileComponents();

    fixture = TestBed.createComponent(HeroComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  }));

  it('should be created', () => {
    expect(component).toBeTruthy();
  });

  it('should log hero name in the console when user submit form', async(() => {
    const heroName = 'Saitama';
    const element = <HTMLFormElement>fixture.debugElement.nativeElement.querySelector('form');

    spyOn(console, 'log').and.callThrough();

    component.form.controls['name'].setValue(heroName);

    element.querySelector('button').click();

    fixture.whenStable().then(() => {
      fixture.detectChanges();
      expect(console.log).toHaveBeenCalledWith(heroName);
    });
  }));

  it('should validate name field as required', () => {

```

```
component.form.controls['name'].setValue('');
expect(component.form.invalid).toBeTruthy();
});
});
```

Nesting components

Components will render in their respective `selector`, so you can use that to nest components.

If you have a component that shows a message:

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-required',
  template: `{{name}} is required.`
})
export class RequiredComponent {
  @Input()
  public name: String = '';
}
```

You can use it inside another component using `app-required` (this component's selector):

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-sample',
  template: `
    <input type="text" name="heroName" />
    <app-required name="Hero Name"></app-required>
  `
})
export class RequiredComponent {
  @Input()
  public name: String = '';
}
```

Read Components online: <https://riptutorial.com/angular2/topic/10838/components>

Chapter 32: Configuring ASP.net Core application to work with Angular 2 and TypeScript

Introduction

SCENARIO: ASP.NET Core background Angular 2 Front-End Angular 2 Components using Asp.net Core Controllers

It way can implement Angular 2 over Asp.Net Core app. It let us call MVC Controllers from Angular 2 components too with the MVC result View supporting Angular 2.

Examples

Asp.Net Core + Angular2 + Gulp

Startup.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using CoreAngular000.Data;
using CoreAngular000.Models;
using CoreAngular000.Services;
using Microsoft.Extensions.FileProviders;
using System.IO;

namespace CoreAngular000
{
    public class Startup
    {
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json", optional: false, reloadOnChange:
true)
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional:
true);

            if (env.IsDevelopment())
            {
```

```

        builder.AddUserSecrets<Startup>();
    }

    builder.AddEnvironmentVariables();
    Configuration = builder.Build();
}

public IConfigurationRoot Configuration { get; }

public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new
PhysicalFileProvider(Path.Combine(env.ContentRootPath, "node_modules"),
        RequestPath = "/node_modules"
    });

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}

```

```
}  
}
```

tsConfig.json

```
{  
  "compilerOptions": {  
    "diagnostics": true,  
    "emitDecoratorMetadata": true,  
    "experimentalDecorators": true,  
    "lib": [ "es2015", "dom" ],  
    "listFiles": true,  
    "module": "commonjs",  
    "moduleResolution": "node",  
    "noImplicitAny": true,  
    "outDir": "wwwroot",  
    "removeComments": false,  
    "rootDir": "wwwroot",  
    "sourceMap": true,  
    "suppressImplicitAnyIndexErrors": true,  
    "target": "es5"  
  },  
  "exclude": [  
    "node_modules",  
    "wwwroot/lib/"  
  ]  
}
```

Package.json

```
{  
  "name": "angular dependencies and web dev package",  
  "version": "1.0.0",  
  "description": "Angular 2 MVC. Samuel Maicas Template",  
  "scripts": {},  
  "dependencies": {  
    "@angular/common": "~2.4.0",  
    "@angular/compiler": "~2.4.0",  
    "@angular/core": "~2.4.0",  
    "@angular/forms": "~2.4.0",  
    "@angular/http": "~2.4.0",  
    "@angular/platform-browser": "~2.4.0",  
    "@angular/platform-browser-dynamic": "~2.4.0",  
    "@angular/router": "~3.4.0",  
    "angular-in-memory-web-api": "~0.2.4",  
    "systemjs": "0.19.40",  
    "core-js": "^2.4.1",  
    "rxjs": "5.0.1",  
    "zone.js": "^0.7.4"  
  },  
  "devDependencies": {  
    "del": "^2.2.2",  
    "gulp": "^3.9.1",  
    "gulp-concat": "^2.6.1",  
    "gulp-cssmin": "^0.1.7",  
    "gulp-htmlmin": "^3.0.0",  
    "gulp-uglify": "^2.1.2",  
    "merge-stream": "^1.0.1",  
    "tslint": "^3.15.1",  
  }  
}
```



```
"typescript": "~2.0.10"
},
"repository": {}
}
```

bundleconfig.json

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

Convert bundleconfig.json to gulpfile (RightClick bundleconfig.json on solution explorer, Bundler&Minifier > Convert to Gulp)

Views/Home/Index.cshtml

```
@{
    ViewData["Title"] = "Home Page";
}
<div>{{ nombre }}</div>
```

For wwwroot folder use <https://github.com/angular/quickstart> seed. You need: **index.html** **main.ts**, **systemjs-angular-loader.js**, **systemjs.config.js**, **tsconfig.json** And the **app** folder

wwwroot/Index.html

```
<html>
<head>
  <title>SMTemplate Angular2 & ASP.NET Core</title>
  <base href="/">
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">

  <script src="node_modules/core-js/client/shim.min.js"></script>

  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>

  <script src="systemjs.config.js"></script>
```

```

    <script>
      System.import('main.js').catch(function(err){ console.error(err); });
    </script>
  </head>

  <body>
    <my-app>Loading AppComponent here ...</my-app>
  </body>
</html>

```

You can call as it to Controllers from templateUrl. wwwroot/app/app.component.ts

```

import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  templateUrl: '/home/index',
})
export class AppComponent { nombre = 'Samuel Maicas'; }

```

[Seed] Asp.Net Core + Angular2 + Gulp on Visual Studio 2017

1. Download seed
2. Run dotnet restore
3. Run npm install

Always. Enjoy.

<https://github.com/SamML/CoreAngular000>

MVC <-> Angular 2

How to: CALL ANGULAR 2 HTML/JS COMPONENT FROM ASP.NET Core CONTROLLER:

We call the HTML instead return View()

```
return File("~/html/About.html", "text/html");
```

And load angular component in the html. Here we can decide if we want to work with same or different module. Depends on situation.

wwwroot/html/About.html

```

<!DOCTYPE html>
<html>
  <head>
    <title>About Page</title>
    <base href="/">
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="../css/site.min.css" rel="stylesheet" type="text/css"/>

    <script src="../node_modules/core-js/client/shim.min.js"></script>

```

```
<script src="../../node_modules/zone.js/dist/zone.js"></script>
<script src="../../node_modules/systemjs/dist/system.src.js"></script>

<script src="../../systemjs.config.js"></script>
<script>
  System.import('../main.js').catch(function(err){ console.error(err); });
</script>
</head>

<body>
  <aboutpage>Loading AppComponent here ...</aboutpage>
</body>
</html>
```

(*)Already this seed needs to load the entire list of resources

How to: CALL ASP.NET Core Controller to show a MVC View with Angular2 support:

```
import { Component } from '@angular/core';

@Component({
  selector: 'aboutpage',
  templateUrl: '/home/about',
})
export class AboutComponent {

}
```

Read [Configuring ASP.net Core application to work with Angular 2 and TypeScript](https://riptutorial.com/angular2/topic/9543/configuring-asp-net-core-application-to-work-with-angular-2-and-typescript) online:
<https://riptutorial.com/angular2/topic/9543/configuring-asp-net-core-application-to-work-with-angular-2-and-typescript>

Chapter 33: Create an Angular 2+ NPM package

Introduction

Sometimes we need to share some component between some apps and publishing it in npm is one of the best ways of doing this.

There are some tricks that we need to know to be able to use a normal component as npm package without changing the structure as inlining external styles.

You can see a minimal example [here](#)

Examples

Simplest package

Here we are sharing some minimal workflow to build and publish an Angular 2+ npm package.

Configuration files

We need some config files to tell `git`, `npm`, `gulp` and `typescript` how to act.

`.gitignore`

First we create a `.gitignore` file to avoid versioning unwanted files and folders. The content is:

```
npm-debug.log
node_modules
jspm_packages
.idea
build
```

`.npmignore`

Second we create a `.npmignore` file to avoid publishing unwanted files and folders. The content is:

```
examples
node_modules
src
```

`gulpfile.js`

We need to create a `gulpfile.js` to tell Gulp how to compile our application. This part is necessary

because we need to minimize and inline all the external templates and styles before publishing our package. The content is:

```
var gulp = require('gulp');
var embedTemplates = require('gulp-angular-embed-templates');
var inlineNg2Styles = require('gulp-inline-ng2-styles');

gulp.task('js:build', function () {
  gulp.src('src/*.ts') // also can use *.js files
    .pipe(embedTemplates({sourceType:'ts'}))
    .pipe(inlineNg2Styles({ base: '/src' }))
    .pipe(gulp.dest('./dist'));
});
```

index.d.ts

The `index.d.ts` file is used by typescript when importing an external module. It helps editor with auto-completion and function tips.

```
export * from './lib';
```

index.js

This is the package entry point. When you install this package using NPM and import in your application, you just need to pass the package name and your application will learn where to find any EXPORTED component of your package.

```
exports.AngularXMinimalNpmPackageModule = require('./lib').AngularXMinimalNpmPackageModule;
```

We used `lib` folder because when we compile our code, the output is placed inside `/lib` folder.

package.json

This file is used to configure your npm publication and defines the necessary packages to it to work.

```
{
  "name": "angular-x-minimal-npm-package",
  "version": "0.0.18",
  "description": "An Angular 2+ Data Table that uses HTTP to create, read, update and delete data from an external API such REST.",
  "main": "index.js",
  "scripts": {
    "watch": "tsc -p src -w",
    "build": "gulp js:build && rm -rf lib && tsc -p dist"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/vinagreti/angular-x-minimal-npm-package.git"
  },
  "keywords": [
    "Angular",
```

```

    "Angular2",
    "Datatable",
    "Rest"
  ],
  "author": "bruno@tzadi.com",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/vinagreti/angular-x-minimal-npm-package/issues"
  },
  "homepage": "https://github.com/vinagreti/angular-x-minimal-npm-package#readme",
  "devDependencies": {
    "gulp": "3.9.1",
    "gulp-angular-embed-templates": "2.3.0",
    "gulp-inline-ng2-styles": "0.0.1",
    "typescript": "2.0.0"
  },
  "dependencies": {
    "@angular/common": "2.4.1",
    "@angular/compiler": "2.4.1",
    "@angular/core": "2.4.1",
    "@angular/http": "2.4.1",
    "@angular/platform-browser": "2.4.1",
    "@angular/platform-browser-dynamic": "2.4.1",
    "rxjs": "5.0.2",
    "zone.js": "0.7.4"
  }
}

```

dist/tsconfig.json

Create a dist folder and place this file inside. This file is used to tell Typescript how to compile your application. Where to get the typescript folder and where to put the compiled files.

```

{
  "compilerOptions": {
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "mapRoot": "",
    "rootDir": ".",
    "target": "es5",
    "lib": ["es6", "es2015", "dom"],
    "inlineSources": true,
    "stripInternal": true,
    "module": "commonjs",
    "moduleResolution": "node",
    "removeComments": true,
    "sourceMap": true,
    "outDir": "../lib",
    "declaration": true
  }
}

```

After create the configuration files, we must create our component and module. This component receives a click and displays a message. It is used like a html tag `<angular-x-minimal-npm-package></angular-x-minimal-npm-package>`. Just instal this npm package and load its module in the model you want to use it.

src/angular-x-minimal-npm-package.component.ts

```
import {Component} from '@angular/core';
@Component({
  selector: 'angular-x-minimal-npm-package',
  styleUrls: ['./angular-x-minimal-npm-package.component.scss'],
  templateUrl: './angular-x-minimal-npm-package.component.html'
})
export class AngularXMinimalNpmPackageComponent {
  message = "Click Me ...";
  onClick() {
    this.message = "Angular 2+ Minimal NPM Package. With external scss and html!";
  }
}
```

src/angular-x-minimal-npm-package.component.html

```
<div>
  <h1 (click)="onClick()">{{message}}</h1>
</div>
```

src/angular-x-data-table.component.css

```
h1{
  color: red;
}
```

src/angular-x-minimal-npm-package.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { AngularXMinimalNpmPackageComponent } from './angular-x-minimal-npm-
package.component';

@NgModule({
  imports: [ CommonModule ],
  declarations: [ AngularXMinimalNpmPackageComponent ],
  exports: [ AngularXMinimalNpmPackageComponent ],
  entryComponents: [ AngularXMinimalNpmPackageComponent ],
})
export class AngularXMinimalNpmPackageModule {}
```

After that, you must compile, build and publish your package.

Build and compile

For build we use `gulp` and for compiling we use `tsc`. The command are set in `package.json` file, at `scripts.build` option. We have this set `gulp js:build && rm -rf lib && tsc -p dist`. This is our chain tasks that will do the job for us.

To build and compile, run the following command at the root of your package:

```
npm run build
```

This will trigger the chain and you will end up with your build in `/dist` folder and the compiled package in your `/lib` folder. This is why in `index.js` we exported the code from `/lib` folder and not from `/src`.

Publish

Now we just need to publish our package so we can install it through npm. For that, just run the command:

```
npm publish
```

That is all!!!

Read [Create an Angular 2+ NPM package online](https://riptutorial.com/angular2/topic/8790/create-an-angular-2plus-npm-package):

<https://riptutorial.com/angular2/topic/8790/create-an-angular-2plus-npm-package>

Chapter 34: Creating an Angular npm library

Introduction

How to publish your NgModule, written in TypeScript in npm registry. Setting up npm project, typescript compiler, rollup and continuous integration build.

Examples

Minimal module with service class

File structure

```
/
  -src/
    awesome.service.ts
    another-awesome.service.ts
    awesome.module.ts
  -index.ts
  -tsconfig.json
  -package.json
  -rollup.config.js
  -.npmignore
```

Service and module

Place your awesome work here.

src/awesome.service.ts:

```
export class AwesomeService {
  public doSomethingAwesome(): void {
    console.log("I am so awesome!");
  }
}
```

src/awesome.module.ts:

```
import { NgModule } from '@angular/core'
import { AwesomeService } from './awesome.service';
import { AnotherAwesomeService } from './another-awesome.service';

@NgModule({
  providers: [AwesomeService, AnotherAwesomeService]
})
export class AwesomeModule {}
```

Make your module and service accessible outside.

/index.ts:

```
export { AwesomeService } from './src/awesome.service';
export { AnotherAwesomeService } from './src/another-awesome.service';
export { AwesomeModule } from './src/awesome.module';
```

Compilation

In `compilerOptions.paths` you need to specify all external modules which you used in your package.

/tsconfig.json

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "declaration": true,
    "stripInternal": true,
    "experimentalDecorators": true,
    "strictNullChecks": false,
    "noImplicitAny": true,
    "module": "es2015",
    "moduleResolution": "node",
    "paths": {
      "@angular/core": ["node_modules/@angular/core"],
      "rxjs/*": ["node_modules/rxjs/*"]
    },
    "rootDir": ".",
    "outDir": "dist",
    "sourceMap": true,
    "inlineSources": true,
    "target": "es5",
    "skipLibCheck": true,
    "lib": [
      "es2015",
      "dom"
    ]
  },
  "files": [
    "index.ts"
  ],
  "angularCompilerOptions": {
    "strictMetadataEmit": true
  }
}
```

Specify your externals again

/rollup.config.js

```
export default {
  entry: 'dist/index.js',
```

```

dest: 'dist/bundles/awesome.module.umd.js',
sourceMap: false,
format: 'umd',
moduleName: 'ng.awesome.module',
globals: {
  '@angular/core': 'ng.core',
  'rxjs': 'Rx',
  'rxjs/Observable': 'Rx',
  'rxjs/ReplaySubject': 'Rx',
  'rxjs/add/operator/map': 'Rx.Observable.prototype',
  'rxjs/add/operator/mergeMap': 'Rx.Observable.prototype',
  'rxjs/add/observable/fromEvent': 'Rx.Observable',
  'rxjs/add/observable/of': 'Rx.Observable'
},
external: ['@angular/core', 'rxjs']
}

```

NPM settings

Now, lets place some instructions for npm

/package.json

```

{
  "name": "awesome-angular-module",
  "version": "1.0.4",
  "description": "Awesome angular module",
  "main": "dist/bundles/awesome.module.umd.min.js",
  "module": "dist/index.js",
  "typings": "dist/index.d.ts",
  "scripts": {
    "test": "",
    "transpile": "ngc",
    "package": "rollup -c",
    "minify": "uglifyjs dist/bundles/awesome.module.umd.js --screw-ie8 --compress --mangle --comments --output dist/bundles/awesome.module.umd.min.js",
    "build": "rimraf dist && npm run transpile && npm run package && npm run minify",
    "prepublishOnly": "npm run build"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/maciejtreder/awesome-angular-module.git"
  },
  "keywords": [
    "awesome",
    "angular",
    "module",
    "minimal"
  ],
  "author": "Maciej Treder <contact@maciejtreder.com>",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/maciejtreder/awesome-angular-module/issues"
  },
  "homepage": "https://github.com/maciejtreder/awesome-angular-module#readme",
  "devDependencies": {
    "@angular/compiler": "^4.0.0",

```

```
  "@angular/compiler-cli": "^4.0.0",
  "rimraf": "^2.6.1",
  "rollup": "^0.43.0",
  "typescript": "^2.3.4",
  "uglify-js": "^3.0.21"
},
"dependencies": {
  "@angular/core": "^4.0.0",
  "rxjs": "^5.3.0"
}
}
```

We can also specify what files, npm should ignore

/.npmignore

```
node_modules
npm-debug.log
Thumbs.db
.DS_Store
src
!dist/src
plugin
!dist/plugin
*.ngsummary.json
*.iml
rollup.config.js
tsconfig.json
*.ts
!*.*.d.ts
.idea
```

Continuous integration

Finally you can set up continuous integration build

.travis.yml

```
language: node_js
node_js:
- node

deploy:
  provider: npm
  email: contact@maciejtreder.com
  api_key:
    secure: <your api key>
  on:
    tags: true
    repo: maciejtreder/awesome-angular-module
```

Demo can be found here: <https://github.com/maciejtreder/awesome-angular-module>

Read **Creating an Angular npm library** online: <https://riptutorial.com/angular2/topic/10704/creating->

Chapter 35: CRUD in Angular2 with Restful API

Syntax

- `@Injectable()` // Tells dependency injector to inject dependencies when creating instance of this service.
- `request.subscribe()` // This is where you *actually* make the request. Without this your request won't be sent. Also you want to read response in the callback function.
- `constructor(private wikiService: WikipediaService) {}` // Since both our service and it's dependencies are injectable by the dependency injector it's a good practice to inject the service to component for unit testing the app.

Examples

Read from an Restful API in Angular2

To separate API logic from the component, we are creating the API client as a separate class. This example class makes a request to Wikipedia API to get random wiki articles.

```
import { Http, Response } from '@angular/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/Rx';

@Injectable()
export class WikipediaService{
  constructor(private http: Http) {}

  getRandomArticles(numberOfArticles: number)
  {
    var request =
this.http.get("https://en.wikipedia.org/w/api.php?action=query&list=random&format=json&rnlimit="
+ numberOfArticles);
    return request.map((response: Response) => {
      return response.json();
    }, (error) => {
      console.log(error);
      //your want to implement your own error handling here.
    });
  }
}
```

And have a component to consume our new API client.

```
import { Component, OnInit } from '@angular/core';
import { WikipediaService } from './wikipedia.Service';
```

```
@Component({
  selector: 'wikipedia',
  templateUrl: 'wikipedia.component.html'
})
export class WikipediaComponent implements OnInit {
  constructor(private wikiService: WikipediaService) { }

  private articles: any[] = null;
  ngOnInit() {
    var request = this.wikiService.getRandomArticles(5);
    request.subscribe((res) => {
      this.articles = res.query.random;
    });
  }
}
```

Read CRUD in Angular2 with Restful API online: <https://riptutorial.com/angular2/topic/7343/crud-in-angular2-with-restful-api>

Chapter 36: custom ngx-bootstrap datepicker + input

Examples

custom ngx-bootstrap datepicker

datepicker.component.html

```
<div (clickOutside)="onClickedOutside($event)" (blur)="onClickedOutside($event)">
  <div class="input-group date" [ngClass]="{'disabled-icon': disabledDatePicker == false}">
    <input (change)="changedDate()" type="text" [ngModel]="value" class="form-control"
    id="{{id}}" (focus)="openCloseDatepicker()" disabled="{{disabledInput}}" />
    <span id="openCloseDatepicker" class="input-group-addon"
    (click)="openCloseDatepicker()">
      <span class="glyphicon-calendar glyphicon"></span>
    </span>
  </div>

  <div class="dp-popup" *ngIf="showDatePicker">
    <datepicker [startingDay]="1" [startingDay]="dt" [minDate]="min" [(ngModel)]="dt"
    (selectionDone)="onSelectionDone($event)"></datepicker>
  </div>
</div>
```

datepicker.component.ts

```
import {Component, Input, EventEmitter, Output, OnChanges, SimpleChanges, ElementRef, OnInit}
from "@angular/core";
import {DatePipe} from "@angular/common";
import {NgModel} from "@angular/forms";
import * as moment from 'moment';

@Component({
  selector: 'custom-datepicker',
  templateUrl: 'datepicker.component.html',
  providers: [DatePipe, NgModel],
  host: {
    '(document:mousedown)': 'onClick($event)',
  }
})

export class DatepickerComponent implements OnChanges , OnInit{
  ngOnInit(): void {
    this.dt = null;
  }

  inputElement : ElementRef;
  dt: Date = null;
  showDatePicker: boolean = false;

  @Input() disabledInput : boolean = false;
```



```

@Input() disabledDatePicker: boolean = false;
@Input() value: string = null;
@Input() id: string;
@Input() min: Date = null;
@Input() max: Date = null;

@Output() dateModelChange = new EventEmitter();
constructor(el: ElementRef) {
  this.inputElement = el;
}

changedDate(){
  if(this.value === ''){
    this.dateModelChange.emit(null);
  }else if(this.value.split('/').length === 3){
    this.dateModelChange.emit(DatepickerComponent.convertToDate(this.value));
  }
}

clickOutside(event : Event){
  if(this.inputElement.nativeElement !== event.target) {
    console.log('click outside', event);
  }
}

onClick(event) {
  if (!this.inputElement.nativeElement.contains(event.target)) {
    this.close();
  }
}

ngOnChanges(changes: SimpleChanges): void {
  if (this.value !== null && this.value !== undefined && this.value.length > 0) {
    this.value = null;
    this.dt = null;
  }else {
    if(this.value !== null){
      this.dt = new Date(this.value);
      this.value = moment(this.value).format('MM/DD/YYYY');
    }
  }
}

private static transformDate(date: Date): string {
  return new DatePipe('pt-PT').transform(date, 'MM/dd/yyyy');
}

openCloseDatepicker(): void {
  if (!this.disabledDatePicker) {
    this.showDatepicker = !this.showDatepicker;
  }
}

open(): void {
  this.showDatepicker = true;
}

close(): void {
  this.showDatepicker = false;
}

private apply(): void {

```

```
    this.value = DatepickerComponent.transformDate(this.dt);
    this.dateModelChange.emit(this.dt);
  }

  onSelectionDone(event: Date): void {
    this.dt = event;
    this.apply();
    this.close();
  }

  onClickedOutside(event: Date): void {
    if (this.showDatepicker) {
      this.close();
    }
  }

  static convertToDate(val : string): Date {
    return new Date(val.replace('/', '-'));
  }
}
```

Read custom ngx-bootstrap datepicker + input online:

<https://riptutorial.com/angular2/topic/10549/custom-ngx-bootstrap-datepicker-plus-input>

Chapter 37: Debugging Angular2 typescript application using Visual Studio Code

Examples

Launch.json setup for you workspace

1. Turn on Debug from menu - view > debug
2. it return some error during start debug, show pop out notification and open launch.json from this popup notification It is just because of launch.json not set for your workspace. copy and paste below code in to launch.json //new launch.json
your old launch.json

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Launch Extension",
      "type": "extensionHost",
      "request": "launch",
      "runtimeExecutable": "${execPath}",
      "args": [
        "--extensionDevelopmentPath=${workspaceRoot}"
      ],
      "stopOnEntry": false,
      "sourceMaps": true,
      "outDir": "${workspaceRoot}/out",
      "preLaunchTask": "npm"
    }
  ]
}
```

Now update your launch.json as below

new launch.json

****// remember please mention your main.js path into it****

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Launch",
      "type": "node",
      "request": "launch",
      "program": "${workspaceRoot}/app/main.js", // put your main.js path
      "stopOnEntry": false,
      "args": [],
      "cwd": "${workspaceRoot}",
      "preLaunchTask": null,
      "runtimeExecutable": null,
    }
  ]
}
```

```

    "runtimeArgs": [
      "--nolazy"
    ],
    "env": {
      "NODE_ENV": "development"
    },
    "console": "internalConsole",
    "sourceMaps": false,
    "outDir": null
  },
  {
    "name": "Attach",
    "type": "node",
    "request": "attach",
    "port": 5858,
    "address": "localhost",
    "restart": false,
    "sourceMaps": false,
    "outDir": null,
    "localRoot": "${workspaceRoot}",
    "remoteRoot": null
  },
  {
    "name": "Attach to Process",
    "type": "node",
    "request": "attach",
    "processId": "${command.PickProcess}",
    "port": 5858,
    "sourceMaps": false,
    "outDir": null
  }
]
}

```

3. Now it debug is working, show notification popup for step by step debugging

Read [Debugging Angular2 typescript application using Visual Studio Code](https://riptutorial.com/angular2/topic/7139/debugging-angular2-typescript-application-using-visual-studio-code) online:

<https://riptutorial.com/angular2/topic/7139/debugging-angular2-typescript-application-using-visual-studio-code>

Chapter 38: Detecting resize events

Examples

A component listening in on the window resize event.

Suppose we have a component which will hide at a certain window width.

```
import { Component } from '@angular/core';

@Component({
  ...
  template: `
    <div>
      <p [hidden]="!visible" (window:resize)="onResize($event)" >Now you see me...</p>
      <p>now you dont!</p>
    </div>
  `
  ...
})
export class MyComponent {
  visible: boolean = false;
  breakpoint: number = 768;

  constructor() {
  }

  onResize(event) {
    const w = event.target.innerWidth;
    if (w >= this.breakpoint) {
      this.visible = true;
    } else {
      // whenever the window is less than 768, hide this component.
      this.visible = false;
    }
  }
}
```

A `p` tag in our template will hide whenever `visible` is false. `visible` will change value whenever the `onResize` event handler is invoked. Its call occurs every time `window:resize` fires an event.

Read Detecting resize events online: <https://riptutorial.com/angular2/topic/5276/detecting-resize-events>

Chapter 39: Directives

Syntax

- `<input [value]="value">` - Binds attribute value class member `name`.
- `<div [attr.data-note]="note">` - Binds attribute `data-note` to variable `note`.
- `<p green></p>` - Custom directive

Remarks

The main source of information about Angular 2 directives is the official documentation <https://angular.io/docs/ts/latest/guide/attribute-directives.html>

Examples

Attribute directive

```
<div [class.active]="isActive"></div>

<span [style.color]='red'></span>

<p [attr.data-note]='This is value for data-note attribute'>A lot of text here</p>
```

Component is a directive with template

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `
    <h1>Angular 2 App</h1>
    <p>Component is directive with template</p>
  `
})
export class AppComponent {
}
```

Structural directives

```
<div *ngFor="let item of items">{{ item.description }}</div>

<span *ngIf="isVisible"></span>
```

Custom directive

```
import {Directive, ElementRef, Renderer} from '@angular/core';

@Directive({
  selector: '[green]',
})

class GreenDirective {
  constructor(private _elementRef: ElementRef,
              private _renderer: Renderer) {
    _renderer.setStyle(_elementRef.nativeElement, 'color', 'green');
  }
}
```

Usage:

```
<p green>A lot of green text here</p>
```

*ngFor

form1.component.ts:

```
import { Component } from '@angular/core';

// Defines example component and associated template
@Component({
  selector: 'example',
  template: `
    <div *ngFor="let f of fruit"> {{f}} </div>
    <select required>
      <option *ngFor="let f of fruit" [value]="f"> {{f}} </option>
    </select>
  `
})

// Create a class for all functions, objects, and variables
export class ExampleComponent {
  // Array of fruit to be iterated by *ngFor
  fruit = ['Apples', 'Oranges', 'Bananas', 'Limes', 'Lemons'];
}
```

Output:

```
<div>Apples</div>
<div>Oranges</div>
<div>Bananas</div>
<div>Limes</div>
<div>Lemons</div>
<select required>
  <option value="Apples">Apples</option>
  <option value="Oranges">Oranges</option>
  <option value="Bananas">Bananas</option>
  <option value="Limes">Limes</option>
  <option value="Lemons">Lemons</option>
</select>
```

In its most simple form, *ngFor has two parts : let **variableName** of **object/array**

In the case of `fruit = ['Apples', 'Oranges', 'Bananas', 'Limes', 'Lemons'];`,

Apples, Oranges, and so on are the values inside the array `fruit`.

`[value]="f"` will be equal to each current `fruit (f)` that `*ngFor` has iterated over.

Unlike AngularJS, Angular2 has not continued with the use of `ng-options` for `<select>` and `ng-repeat` for all other general repetitions.

`*ngFor` is very similar to `ng-repeat` with slightly varied syntax.

References:

Angular2 | [Displaying Data](#)

Angular2 | [ngFor](#)

Angular2 | [Forms](#)

Copy to Clipboard directive

In this example we are going to create a directive to copy a text into the clipboard by clicking on an element

copy-text.directive.ts

```
import {
  Directive,
  Input,
  HostListener
} from "@angular/core";

@Directive({
  selector: '[text-copy]'
})
export class TextCopyDirective {

  // Parse attribute value into a 'text' variable
  @Input('text-copy') text:string;

  constructor() {
  }

  // The HostListener will listen to click events and run the below function, the
  // HostListener supports other standard events such as mouseenter, mouseleave etc.
  @HostListener('click') copyText() {

    // We need to create a dummy textarea with the text to be copied in the DOM
    var textArea = document.createElement("textarea");

    // Hide the textarea from actually showing
    textArea.style.position = 'fixed';
    textArea.style.top = '-999px';
    textArea.style.left = '-999px';
```



```

    textArea.style.width = '2em';
    textArea.style.height = '2em';
    textArea.style.padding = '0';
    textArea.style.border = 'none';
    textArea.style.outline = 'none';
    textArea.style.boxShadow = 'none';
    textArea.style.background = 'transparent';

    // Set the texarea's content to our value defined in our [text-copy] attribute
    textArea.value = this.text;
    document.body.appendChild(textArea);

    // This will select the textarea
    textArea.select();

    try {
        // Most modern browsers support execCommand('copy'|'cut'|'paste'), if it doesn't
it should throw an error
        var successful = document.execCommand('copy');
        var msg = successful ? 'successful' : 'unsuccessful';
        // Let the user know the text has been copied, e.g toast, alert etc.
        console.log(msg);
    } catch (err) {
        // Tell the user copying is not supported and give alternative, e.g alert window
with the text to copy
        console.log('unable to copy');
    }

    // Finally we remove the textarea from the DOM
    document.body.removeChild(textArea);
}
}

export const TEXT_COPY_DIRECTIVES = [TextCopyDirective];

```

some-page.component.html

Remember to inject TEXT_COPY_DIRECTIVES into the directives array of your component

```

...
<!-- Insert variable as the attribute's value, let textToBeCopied = 'http://facebook.com/'
-->
<button [text-copy]="textToBeCopied">Copy URL</button>
<button [text-copy]=" 'https://www.google.com/' ">Copy URL</button>
...

```

Testing a custom directive

Given a directive that highlights text on mouse events

```

import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({ selector: '[appHighlight]' })
export class HighlightDirective {
    @Input('appHighlight') // tslint:disable-line no-input-rename
    highlightColor: string;

```

```

constructor(private el: ElementRef) { }

@HostListener('mouseenter')
onMouseEnter() {
  this.highlight(this.highlightColor || 'red');
}

@HostListener('mouseleave')
onMouseLeave() {
  this.highlight(null);
}

private highlight(color: string) {
  this.el.nativeElement.style.backgroundColor = color;
}
}

```

It can be tested like this

```

import { ComponentFixture, ComponentFixtureAutoDetect, TestBed } from '@angular/core/testing';

import { Component } from '@angular/core';
import { HighlightDirective } from './highlight.directive';

@Component({
  selector: 'app-test-container',
  template: `
    <div>
      <span id="red" appHighlight>red text</span>
      <span id="green" [appHighlight]="'green'">green text</span>
      <span id="no">no color</span>
    </div>
  `
})
class ContainerComponent { }

const mouseEvents = {
  get enter() {
    const mouseenter = document.createEvent('MouseEvent');
    mouseenter.initEvent('mouseenter', true, true);
    return mouseenter;
  },
  get leave() {
    const mouseleave = document.createEvent('MouseEvent');
    mouseleave.initEvent('mouseleave', true, true);
    return mouseleave;
  },
};

describe('HighlightDirective', () => {
  let fixture: ComponentFixture<ContainerComponent>;
  let container: ContainerComponent;
  let element: HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ContainerComponent, HighlightDirective],
      providers: [
        { provide: ComponentFixtureAutoDetect, useValue: true },
      ],
    });
  });
});

```

```

});

fixture = TestBed.createComponent(ContainerComponent);
// fixture.detectChanges(); // without the provider
container = fixture.componentInstance;
element = fixture.nativeElement;
});

it('should set background-color to empty when mouse leaves with directive without
arguments', () => {
  const targetElement = <HTMLSpanElement>element.querySelector('#red');

  targetElement.dispatchEvent(mouseEvents.leave);
  expect(targetElement.style.backgroundColor).toEqual('');
});

it('should set background-color to empty when mouse leaves with directive with arguments',
() => {
  const targetElement = <HTMLSpanElement>element.querySelector('#green');

  targetElement.dispatchEvent(mouseEvents.leave);
  expect(targetElement.style.backgroundColor).toEqual('');
});

it('should set background-color red with no args passed', () => {
  const targetElement = <HTMLSpanElement>element.querySelector('#red');

  targetElement.dispatchEvent(mouseEvents.enter);
  expect(targetElement.style.backgroundColor).toEqual('red');
});

it('should set background-color green when passing green parameter', () => {
  const targetElement = <HTMLSpanElement>element.querySelector('#green');

  targetElement.dispatchEvent(mouseEvents.enter);
  expect(targetElement.style.backgroundColor).toEqual('green');
});
});

```

Read Directives online: <https://riptutorial.com/angular2/topic/2202/directives>

Chapter 40: Directives & components :

@Input @Output

Syntax

1. One way binding from parent component to nested component: [propertyName]
2. One way binding from nested component to parent component: (propertyName)
3. Two-way binding (a.k.a banana box notation) : [(propertyName)]

Examples

Input example

@input is useful to bind data between components

First, import it in your component

```
import { Input } from '@angular/core';
```

Then, add the input as a property of your component class

```
@Input() car: any;
```

Let's say that the selector of your component is 'car-component', when you call the component, add the attribute 'car'

```
<car-component [car]="car"></car-component>
```

Now your car is accessible as an attribute in your object (this.car)

Full Example :

1. car.entity.ts

```
export class CarEntity {  
  constructor(public brand : string, public color : string) {  
  }  
}
```

2. car.component.ts

```
import { Component, Input } from '@angular/core';  
import { CarEntity } from "../car.entity";  
  
@Component({
```

```

    selector: 'car-component',
    template: require('./templates/car.html'),
  })

export class CarComponent {
  @Input() car: CarEntity;

  constructor() {
    console.log('gros');
  }
}

```

3. garage.component.ts

```

import { Component } from '@angular/core';
import { CarEntity } from "../car.entity";
import { CarComponent } from "../car.component";

@Component({
  selector: 'garage',
  template: require('./templates/garage.html'),
  directives: [CarComponent]
})

export class GarageComponent {
  public cars : Array<CarEntity>;

  constructor() {
    var carOne : CarEntity = new CarEntity('renault', 'blue');
    var carTwo : CarEntity = new CarEntity('fiat', 'green');
    var carThree : CarEntity = new CarEntity('citroen', 'yellow');
    this.cars = [carOne, carTwo, carThree];
  }
}

```

4. garage.html

```

<div *ngFor="let car of cars">
  <car-component [car]="car"></car-component>
</div>

```

5. car.html

```

<div>
  <span>{{ car.brand }}</span> |
  <span>{{ car.color }}</span>
</div>

```

Angular2 @Input and @Output in a nested component

A Button directive which accepts an @Input() to specify a click limit until the button gets disabled. The parent component can listen to an event which will be emitted when the click limit is reached via @Output:

```

import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'limited-button',
  template: `<button (click)="onClick()"
              [disabled]="disabled">
              <ng-content></ng-content>
            </button>`,
  directives: []
})

export class LimitedButton {
  @Input() clickLimit: number;
  @Output() limitReached: EventEmitter<number> = new EventEmitter();

  disabled: boolean = false;

  private clickCount: number = 0;

  onClick() {
    this.clickCount++;
    if (this.clickCount === this.clickLimit) {
      this.disabled = true;
      this.limitReached.emit(this.clickCount);
    }
  }
}

```

Parent component which uses the Button directive and alerts a message when the click limit is reached:

```

import { Component } from '@angular/core';
import { LimitedButton } from './limited-button.component';

@Component({
  selector: 'my-parent-component',
  template: `<limited-button [clickLimit]="2"
                          (limitReached)="onLimitReached($event)">
              You can only click me twice
            </limited-button>`,
  directives: [LimitedButton]
})

export class MyParentComponent {
  onLimitReached(clickCount: number) {
    alert('Button disabled after ' + clickCount + ' clicks.');
```

Angular2 @Input with asynchronous data

Sometimes you need to fetch data asynchronously before passing it to a child component to use. If the child component tries to use the data before it has been received, it will throw an error. You can use `ngOnChanges` to detect changes in a components' `@Inputs` and wait until they are defined before acting upon them.

Parent component with async call to an endpoint

```
import { Component, OnChanges, OnInit } from '@angular/core';
import { Http, Response } from '@angular/http';
import { ChildComponent } from './child.component';

@Component ({
  selector : 'parent-component',
  template : `
    <child-component [data]="asyncData"></child-component>
  `
})
export class ParentComponent {

  asyncData : any;

  constructor(
    private _http : Http
  ){}

  ngOnInit () {
    this._http.get('some.url')
      .map(this.extractData)
      .subscribe(this.handleData)
      .catch(this.handleError);
  }

  extractData (res:Response) {
    let body = res.json();
    return body.data || { };
  }

  handleData (data:any) {
    this.asyncData = data;
  }

  handleError (error:any) {
    console.error(error);
  }
}
```

Child component which has async data as input

This child component takes the async data as input. Therefore it must wait for the data to exist before Using it. We use `ngOnChanges` which fires whenever a component's input changes, check if the data exists and use it if it does. Notice that the template for the child will not show if a property that relies on the data being passed in is not true.

```
import { Component, OnChanges, Input } from '@angular/core';
```

```
@Component ({
  selector : 'child-component',
  template : `
    <p *ngIf="doesDataExist">Hello child</p>
  `
})
export class ChildComponent {

  doesDataExist: boolean = false;

  @Input('data') data : any;

  // Runs whenever component @Inputs change
  ngOnChanges () {
    // Check if the data exists before using it
    if (this.data) {
      this.useData(data);
    }
  }

  // contrived example to assign data to reliesOnData
  useData (data) {
    this.doesDataExist = true;
  }
}
```

Read Directives & components : @Input @Output online:

<https://riptutorial.com/angular2/topic/3046/directives---components----input--output>

Chapter 41: Dropzone in Angular2

Examples

Dropzone

Angular 2 wrapper library for Dropzone.

```
npm install angular2-dropzone-wrapper --save-dev
```

Load the module for your app-module

```
import { DropzoneModule } from 'angular2-dropzone-wrapper';
import { DropzoneConfigInterface } from 'angular2-dropzone-wrapper';

const DROPZONE_CONFIG: DropzoneConfigInterface = {
  // Change this to your upload POST address:
  server: 'https://example.com/post',
  maxFileSize: 10,
  acceptedFiles: 'image/*'
};

@NgModule({
  ...
  imports: [
    ...
    DropzoneModule.forRoot(DROPZONE_CONFIG)
  ]
})
```

COMPONENT USAGE

Simply replace the element that would ordinarily be passed to Dropzone with the dropzone component.

```
<dropzone [config]="config" [message]='Click or drag images here to upload'
(error)="onUploadError($event)" (success)="onUploadSuccess($event)"></dropzone>
```

Create dropzone component

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-new-media',
  templateUrl: './dropzone.component.html',
  styleUrls: ['./dropzone.component.scss']
})
export class DropZoneComponent {

  onUploadError(args: any) {
    console.log('onUploadError:', args);
  }
}
```

```
onUploadSuccess(args: any) {  
    console.log('onUploadSuccess:', args);  
}  
}
```

Read Dropzone in Angular2 online: <https://riptutorial.com/angular2/topic/10010/dropzone-in-angular2>

Chapter 42: Dynamically add components using `ViewContainerRef.createComponent`

Examples

A wrapper component that adds dynamic components declaratively

A custom component that takes the type of a component as input and creates an instance of that component type inside itself. When the input is updated, the previously added dynamic component is removed and the new one added instead.

```
@Component({
  selector: 'dcl-wrapper',
  template: `<div #target></div>`
})
export class DclWrapper {
  @ViewChild('target', {
    read: ViewContainerRef
  }) target;
  @Input() type;
  cmpRef: ComponentRef;
  private isViewInitialized: boolean = false;

  constructor(private resolver: ComponentResolver) {}

  updateComponent() {
    if (!this.isViewInitialized) {
      return;
    }
    if (this.cmpRef) {
      this.cmpRef.destroy();
    }
    this.resolver.resolveComponent(this.type).then((factory: ComponentFactory < any > ) => {
      this.cmpRef = this.target.createComponent(factory)
      // to access the created instance use
      // this.cmpRef.instance.someProperty = 'someValue';
      // this.cmpRef.instance.someOutput.subscribe(val => doSomething());
    });
  }

  ngOnChanges() {
    this.updateComponent();
  }

  ngAfterViewInit() {
    this.isViewInitialized = true;
    this.updateComponent();
  }

  ngOnDestroy() {
    if (this.cmpRef) {
      this.cmpRef.destroy();
    }
  }
}
```

```
}
```

This allows you to create dynamic components like

```
<dcl-wrapper [type]="someComponentType"></dcl-wrapper>
```

Plunker example

Dynamically add component on specific event(click)

Main Component File:

```
//our root app component
import {Component, NgModule, ViewChild, ViewContainerRef, ComponentFactoryResolver,
ComponentRef} from '@angular/core'
import {BrowserModule} from '@angular/platform-browser'
import {ChildComponent} from './childComp.ts'

@Component({
  selector: 'my-app',
  template: `
    <div>
      <h2>Hello {{name}}</h2>
      <input type="button" value="Click me to add element" (click) = addElement()> // call the
function on click of the button
      <div #parent> </div> // Dynamic component will be loaded here
    </div>
  `,
})
export class App {
  name:string;

  @ViewChild('parent', {read: ViewContainerRef}) target: ViewContainerRef;
  private componentRef: ComponentRef<any>;

  constructor(private componentFactoryResolver: ComponentFactoryResolver) {
    this.name = 'Angular2'
  }

  addElement(){
    let childComponent =
this.componentFactoryResolver.resolveComponentFactory(ChildComponent);
    this.componentRef = this.target.createComponent(childComponent);
  }
}
```

childComp.ts :

```
import{Component} from '@angular/core';

@Component({
  selector: 'child',
  template: `
    <p>This is Child</p>
  `,
})
```

```
export class ChildComponent {
  constructor(){

  }
}
```

app.module.ts :

```
@NgModule({
  imports: [ BrowserModule ],
  declarations: [ App, ChildComponent ],
  bootstrap: [ App ],
  entryComponents: [ChildComponent] // define the dynamic component here in module.ts
})
export class AppModule {}
```

Plunker example

Rendered dynamically created component array on template html in Angular2

We can create dynamic component and get the instances of component into an array and finally rendered it on template.

For example, we can consider two widget component, ChartWidget and PatientWidget which extended the class WidgetComponent that I wanted to add in the container.

ChartWidget.ts

```
@Component({
  selector: 'chart-widget',
  templateUrl: 'chart-widget.component.html',
  providers: [{provide: WidgetComponent, useExisting: forwardRef(() => ChartWidget) }]
})

export class ChartWidget extends WidgetComponent implements OnInit {
  constructor(ngEl: ElementRef, renderer: Renderer) {
    super(ngEl, renderer);
  }
  ngOnInit() {}
  close(){
    console.log('close');
  }
  refresh(){
    console.log('refresh');
  }
  ...
}
```

chart-widget.component.html (using primeng Panel)

```
<p-panel [style]="{'margin-bottom':'20px'}">
  <p-header>
    <div class="ui-helper-clearfix">
      <span class="ui-panel-title" style="font-size:14px;display:inline-block;margin-top:2px">Chart Widget</span>
```

```

        <div class="ui-toolbar-group-right">
            <button pButton type="button" icon="fa-window-minimize"
(click)="minimize()" ></button>
            <button pButton type="button" icon="fa-refresh" (click)="refresh()" ></button>
            <button pButton type="button" icon="fa-expand" (click)="expand()" ></button>
            <button pButton type="button" (click)="close()" icon="fa-window-close"></button>
        </div>
    </div>
</p-header>
    some data
</p-panel>

```

DataWidget.ts

```

@Component({
  selector: 'data-widget',
  templateUrl: 'data-widget.component.html',
  providers: [{provide: WidgetComponent, useExisting: forwardRef(() =>DataWidget) }]
})

export class DataWidget extends WidgetComponent implements OnInit {
  constructor(ngEl: ElementRef, renderer: Renderer) {
    super(ngEl, renderer);
  }
  ngOnInit() {}
  close(){
    console.log('close');
  }
  refresh(){
    console.log('refresh');
  }
  ...
}

```

data-widget.component.html (same as chart-widget using primeng Panel)

WidgetComponent.ts

```

@Component({
  selector: 'widget',
  template: '<ng-content></ng-content>'
})
export class WidgetComponent{
}

```

we can creat dynamic component instances by selecting the pre-existing components. For example,

```

@Component({
  selector: 'dynamic-component',
  template: `<div #container><ng-content></ng-content></div>`
})
export class DynamicComponent {
  @ViewChild('container', {read: ViewContainerRef}) container: ViewContainerRef;
}

```

```

    public addComponent(ngItem: Type<WidgetComponent>): WidgetComponent {
    let factory = this.compFactoryResolver.resolveComponentFactory(ngItem);
    const ref = this.container.createComponent(factory);
    const newItem: WidgetComponent = ref.instance;
    this._elements.push(newItem);
    return newItem;
    }
}

```

Finally we use it in app component. app.component.ts

```

@Component({
  selector: 'app-root',
  templateUrl: './app/app.component.html',
  styleUrls: ['./app/app.component.css'],
  entryComponents: [ChartWidget, DataWidget],
})

export class AppComponent {
  private elements: Array<WidgetComponent>=[];
  private WidgetClasses = {
    'ChartWidget': ChartWidget,
    'DataWidget': DataWidget
  }
  @ViewChild(DynamicComponent) dynamicComponent:DynamicComponent;

  addComponent(widget: string ): void{
    let ref= this.dynamicComponent.addComponent(this.WidgetClasses[widget]);
    this.elements.push(ref);
    console.log(this.elements);

    this.dynamicComponent.resetContainer();
  }
}

```

app.component.html

```

<button (click)="addComponent('ChartWidget')">Add ChartWidget</button>
<button (click)="addComponent('DataWidget')">Add DataWidget</button>

<dynamic-component [hidden]="true" ></dynamic-component>

<hr>
Dynamic Components
<hr>
<widget *ngFor="let item of elements">
  <div>{{item}}</div>
  <div [innerHTML]="item._ngEl.nativeElement.innerHTML | sanitizeHtml">
  </div>
</widget>

```

<https://plnkr.co/edit/lugU2pPsSBd3XhPHiUP1?p=preview>

Some modification by @yurzui to use mouse event on the widgets

view.directive.ts

```
import { ViewRef, Directive, Input, ViewContainerRef } from '@angular/core';
```

```
@Directive({
  selector: '[view]'
})
export class ViewDirective {
  constructor(private vcRef: ViewContainerRef) {}

  @Input()
  set view(view: ViewRef) {
    this.vcRef.clear();
    this.vcRef.insert(view);
  }

  ngOnDestroy() {
    this.vcRef.clear();
  }
}
```

app.component.ts

```
private elements: Array<{ view: ViewRef, component: WidgetComponent}> = [];

...
addComponent(widget: string ): void{
  let component = this.dynamicComponent.addComponent(this.WidgetClasses[widget]);
  let view: ViewRef = this.dynamicComponent.container.detach(0);
  this.elements.push({view, component});

  this.dynamicComponent.resetContainer();
}
```

app.component.html

```
<widget *ngFor="let item of elements">
  <ng-container *view="item.view"></ng-container>
</widget>
```

<https://plnkr.co/edit/JHpIHR43SvJd0OxJVMfV?p=preview>

Read Dynamically add components using ViewContainerRef.createComponent online:

<https://riptutorial.com/angular2/topic/831/dynamically-add-components-using-viewcontainerref-createcomponent>

Chapter 43: EventEmitter Service

Examples

Class Overview

```
class EventEmitter extends Subject {
  constructor(isAsync?: boolean)
  emit(value?: T)
  subscribe(generatorOrNext?: any, error?: any, complete?: any) : any
}
```

Class Component

```
@Component({
  selector: 'zippy',
  template: `
<div class="zippy">
  <div (click)="toggle()">Toggle</div>
  <div [hidden]="!visible">
    <ng-content></ng-content>
  </div>
</div>`})
export class Zippy {
  visible: boolean = true;
  @Output() open: EventEmitter<any> = new EventEmitter();
  @Output() close: EventEmitter<any> = new EventEmitter();
  toggle() {
    this.visible = !this.visible;
    if (this.visible) {
      this.open.emit(null);
    } else {
      this.close.emit(null);
    }
  }
}
```

Emmiting Events

```
<zippy (open)="onOpen($event)" (close)="onClose($event)"></zippy>
```

Catching the event

Create a service-

```
import {EventEmitter} from 'angular2/core';
export class NavService {
  navchange: EventEmitter<number> = new EventEmitter();
  constructor() {}
  emitNavChangeEvent(number) {
    this.navchange.emit(number);
  }
}
```

```

    }
    getNavChangeEmitter() {
        return this.navchange;
    }
}

```

Create a component to use the service-

```

import {Component} from 'angular2/core';
import {NavService} from '../services/NavService';

@Component({
  selector: 'obs-comp',
  template: `obs component, item: {{item}}`
})
export class ObservingComponent {
  item: number = 0;
  subscription: any;
  constructor(private navService:NavService) {}
  ngOnInit() {
    this.subscription = this.navService.getNavChangeEmitter()
      .subscribe(item => this.selectedNavItem(item));
  }
  selectedNavItem(item: number) {
    this.item = item;
  }
  ngOnDestroy() {
    this.subscription.unsubscribe();
  }
}

@Component({
  selector: 'my-nav',
  template: `
    <div class="nav-item" (click)="selectedNavItem(1)">nav 1 (click me)</div>
    <div class="nav-item" (click)="selectedNavItem(2)">nav 2 (click me)</div>
  `
})
export class Navigation {
  item = 1;
  constructor(private navService:NavService) {}
  selectedNavItem(item: number) {
    console.log('selected nav item ' + item);
    this.navService.emitNavChangeEvent(item);
  }
}

```

Live example

A live example for this can be found [here](#).

Read EventEmitter Service online: <https://riptutorial.com/angular2/topic/9159/eventemitter-service>

Chapter 44: Example for routes such as /route/subroute for static urls

Examples

Basic route example with sub routes tree

app.module.ts

```
import {routes} from "./app.routes";

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, mainModule.forRoot(), RouterModule.forRoot(routes)],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

app.routes.ts

```
import { Routes } from '@angular/router';
import {SubTreeRoutes} from "./subTree/subTreeRoutes.routes";

export const routes: Routes = [
  ...SubTreeRoutes,
  { path: '', redirectTo: 'home', pathMatch: 'full' }
];
```

subTreeRoutes.ts

```
import {Route} from '@angular/router';
import {exampleComponent} from "./example.component";

export const SubTreeRoutes: Route[] = [
  {
    path: 'subTree',
    children: [
      {path: '', component: exampleComponent}
    ]
  }
];
```

Read Example for routes such as /route/subroute for static urls online:

<https://riptutorial.com/angular2/topic/8910/example-for-routes-such-as--route-subroute-for-static-urls>

Chapter 45: Feature Modules

Examples

A Feature Module

```
// my-feature.module.ts
import { CommonModule } from '@angular/common';
import { NgModule }      from '@angular/core';

import { MyComponent } from './my.component';
import { MyDirective } from './my.directive';
import { MyPipe }      from './my.pipe';
import { MyService }   from './my.service';

@NgModule({
  imports:      [ CommonModule ],
  declarations: [ MyComponent, MyDirective, MyPipe ],
  exports:     [ MyComponent ],
  providers:   [ MyService ]
})
export class MyFeatureModule { }
```

Now, in your root (usually `app.module.ts`):

```
// app.module.ts
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }  from './app.component';
import { MyFeatureModule } from './my-feature.module';

@NgModule({
  // import MyFeatureModule in root module
  imports:      [ BrowserModule, MyFeatureModule ],
  declarations: [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

Read Feature Modules online: <https://riptutorial.com/angular2/topic/6551/feature-modules>

Chapter 46: How to use ngfor

Introduction

The `ngFor` directive is used by Angular2 to instantiate a template once for every item in an iterable object. This directive binds the iterable to the DOM, so if the content of the iterable changes, the content of the DOM will be also changed.

Examples

Unordered list example

```
<ul>
  <li *ngFor="let item of items">{{item.name}}</li>
</ul>
```

More complex template example

```
<div *ngFor="let item of items">
  <p>{{item.name}}</p>
  <p>{{item.price}}</p>
  <p>{{item.description}}</p>
</div>
```

Tracking current interaction example

```
<div *ngFor="let item of items; let i = index">
  <p>Item number: {{i}}</p>
</div>
```

In this case, `i` will take the value of `index`, which is the current loop iteration.

Angular2 aliased exported values

Angular2 provides several exported values that can be aliased to local variables. These are:

- `index`
- `first`
- `last`
- `even`
- `odd`

Except `index`, the other ones take a `Boolean` value. As the previous example using `index`, it can be used any of these exported values:

```
<div *ngFor="let item of items; let firstItem = first; let lastItem = last">
  <p *ngIf="firstItem">I am the first item and I am gonna be showed</p>
  <p *ngIf="firstItem">I am not the first item and I will not show up :(</p>
  <p *ngIf="lastItem">But I'm gonna be showed as I am the last item :)</p>
</div>
```

*ngFor with pipe

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'even'
})

export class EvenPipe implements PipeTransform {
  transform(value: string): string {
    if(value && value %2 === 0){
      return value;
    }
  }
}

@Component({
  selector: 'example-component',
  template: '<div>
    <div *ngFor="let number of numbers | even">
      {{number}}
    </div>
  </div>'
})

export class exampleComponent {
  let numbers : List<number> = Array.apply(null, {length: 10}).map(Number.call, Number)
}
```

Read How to use ngfor online: <https://riptutorial.com/angular2/topic/8051/how-to-use-ngfor>

Chapter 47: How to Use ngIf

Introduction

***NgIf:** It removes or recreates a part of DOM tree depending on an expression evaluation. It is a structural directive and structural directives alter the layout of the DOM by adding, replacing and removing its elements.

Syntax

- `<div *ngIf="false"> test </div> <!-- evaluates to false -->`
- `<div *ngIf="undefined"> test </div> <!-- evaluates to false -->`
- `<div *ngIf="null"> test </div> <!-- evaluates to false -->`
- `<div *ngIf="0"> test </div> <!-- evaluates to false -->`
- `<div *ngIf="NaN"> test </div> <!-- evaluates to false -->`
- `<div *ngIf=""> test </div> <!-- evaluates to false -->`
- All other values evaluate to true.

Examples

Display a loading message

If our component is not ready and waiting for data from server, then we can add loader using `*ngIf`.

Steps:

First declare a boolean:

```
loading: boolean = false;
```

Next, in your component add a lifecycle hook called `ngOnInit`

```
ngOnInit() {  
  this.loading = true;  
}
```

and after you get complete data from server set you loading boolean to false.

```
this.loading=false;
```

In your html template use `*ngIf` with the `loading` property:

```
<div *ngIf="loading" class="progress">  
  <div class="progress-bar info" style="width: 125%;"></div>  
</div>
```

Show Alert Message on a condition

```
<p class="alert alert-success" *ngIf="names.length > 2">Currently there are more than 2 names!</p>
```

To run a function at the start or end of *ngFor loop Using *ngIf

NgFor provides Some values that can be aliased to local variables

- **index** -(variable) position of the current item in the iterable starting at 0
- **first** -(boolean) true if the current item is the first item in the iterable
- **last** -(boolean) true if the current item is the last item in the iterable
- **even** -(boolean) true if the current index is an even number
- **odd** -(boolean) true if the current index is an odd number

```
<div *ngFor="let note of csvdata; let i=index; let lastcall=last">
  <h3>{{i}}</h3> <!-- to show index position
  <h3>{{note}}</h3>
  <span *ngIf="lastcall">{{anyfunction()}} </span><!-- this lastcall boolean value will be
true only if this is last in loop
  // anyfunction() will run at the end of loop same way we can do at start
</div>
```

Use *ngIf with*ngFor

While you are not allowed to use `*ngIf` and `*ngFor` in the same div (it will gives an error in the runtime) you can nest the `*ngIf` in the `*ngFor` to get the desired behavior.

Example 1: General syntax

```
<div *ngFor="let item of items; let i = index">
  <div *ngIf="<your condition here>">

    <!-- Execute code here if statement true -->

  </div>
</div>
```

Example 2: Display elements with even index

```
<div *ngFor="let item of items; let i = index">
  <div *ngIf="i % 2 == 0">
    {{ item }}
  </div>
</div>
```

The downside is that an additional outer `div` element needs to be added.

But consider this use case where a `div` element needs to be iterated (using `*ngFor`) and also includes a check whether the element need to be removed or not (using `*ngIf`), but adding an

additional `div` is not preferred. In this case you can use the `template` tag for the `*ngFor`:

```
<template ngFor let-item [ngForOf]="items">
  <div *ngIf="item.price > 100">
  </div>
</template>
```

This way adding an additional outer `div` is not needed and furthermore the `<template>` element won't be added to the DOM. The only elements added in the DOM from the above example are the iterated `div` elements.

Note: In Angular v4 `<template>` has been deprecated in favour of `<ng-template>` and will be removed in v5. In Angular v2.x releases `<template>` is still valid.

Read How to Use `ngif` online: <https://riptutorial.com/angular2/topic/8346/how-to-use-ngif>

Chapter 48: Http Interceptor

Remarks

What we do with `HttpServiceLayer` class is extend the `Http` class from angular and add our own logic to it.

We then inject that class in the bootstrap class of the application and tell angular that were we import the `Http` class, in the back to insert the `HttpServiceLayer`.

Anywhere in the code we can simply import

```
import { Http } from '@angular/http';
```

But our class will be used for each call.

Examples

Simple Class Extending angular's Http class

```
import { Http, Request, RequestOptionsArgs, Response, RequestOptions, ConnectionBackend, Headers } from '@angular/http';
import { Router } from '@angular/router';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/empty';
import 'rxjs/add/observable/throw';
import 'rxjs/add/operator/catch';
import { ApplicationConfiguration } from '../application-configuration/application-configuration';

/**
 * This class extends the Http class from angular and adds automatically the server URL(if in
 * development mode) and 2 headers by default:
 * Headers added: 'Content-Type' and 'X-AUTH-TOKEN'.
 * 'Content-Type' can be set in any othe service, and if set, it will NOT be overwritten in
 * this class any more.
 */
export class HttpServiceLayer extends Http {

    constructor(backend: ConnectionBackend, defaultOptions: RequestOptions, private _router: Router, private appConfig: ApplicationConfiguration) {
        super(backend, defaultOptions);
    }

    request(url: string | Request, options?: RequestOptionsArgs): Observable<Response> {
        this.getRequestOptionArgs(options);
        return this.intercept(super.request(this.appConfig.getServerAdress() + url, options));
    }

    /**
     * This method checks if there are any headers added and if not created the headers map and
     * ads 'Content-Type' and 'X-AUTH-TOKEN'
     */
}
```

```

* 'Content-Type' is not overwritten if it is already available in the headers map
*/
getRequestOptionArgs(options?: RequestOptionsArgs): RequestOptionsArgs {
  if (options == null) {
    options = new RequestOptions();
  }
  if (options.headers == null) {
    options.headers = new Headers();
  }

  if (!options.headers.get('Content-Type')) {
    options.headers.append('Content-Type', 'application/json');
  }

  if (this.appConfig.getAuthToken() != null) {
    options.headers.append('X-AUTH-TOKEN', this.appConfig.getAuthToken());
  }

  return options;
}

/**
 * This method as the name suggests intercepts the request and checks if there are any errors.
 * If an error is present it will be checked what error there is and if it is a general one
then it will be handled here, otherwise, will be
 * thrown up in the service layers
 */
intercept(observable: Observable<Response>): Observable<Response> {

  // return observable;
  return observable.catch((err, source) => {
    if (err.status == 401) {
      this._router.navigate(['/login']);
      //return observable;
      return Observable.empty();
    } else {
      //return observable;
      return Observable.throw(err);
    }
  });
}
}

```

Using our class instead of Angular's Http

After extending the Http class, we need to tell angular to use this class instead of Http class.

In order to do this, in our main module(or depending on the needs, just a particular module), we need to write in the providers section:

```

export function httpServiceFactory(xhrBackend: XHRBackend, requestOptions: RequestOptions,
router: Router, appConfig: ApplicationConfiguration) {
  return new HttpServiceLayer(xhrBackend, requestOptions, router, appConfig);
}

import { HttpModule, Http, Request, RequestOptionsArgs, Response, XHRBackend, RequestOptions,
ConnectionBackend, Headers } from '@angular/http';
import { Router } from '@angular/router';

```

```

@NgModule({
  declarations: [ ... ],
  imports: [ ... ],
  exports: [ ... ],
  providers: [
    ApplicationConfiguration,
    {
      provide: Http,
      useFactory: httpServiceFactory,
      deps: [XHRBackend, RequestOptions, Router, ApplicationConfiguration]
    }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Note: ApplicationConfiguration is just a service I use to hold some values for the duration of the application

Simple HttpClient AuthToken Interceptor (Angular 4.3+)

```

import { Injectable } from '@angular/core';
import { HttpEvent, HttpRequest, HttpInterceptor, HttpHandler } from '@angular/common/http';
import { UserService } from '../services/user.service';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class AuthHeaderInterceptor implements HttpInterceptor {

  constructor(private userService: UserService) {
  }

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    if (this.userService.isAuthenticated()) {
      req = req.clone({
        setHeaders: {
          Authorization: `Bearer ${this.userService.token}`
        }
      });
    }
    return next.handle(req);
  }
}

```

Providing Interceptor (some-module.module.ts)

```
{provide: HTTP_INTERCEPTORS, useClass: AuthHeaderInterceptor, multi: true},
```

Read Http Interceptor online: <https://riptutorial.com/angular2/topic/1413/http-interceptor>

Chapter 49: Installing 3rd party plugins with angular-cli@1.0.0-beta.10

Remarks

It is possible to install other libraries following, this approach, however, there might be a need to specify the module type, main file, and default extension.

```
'lodash': {  
  format: 'cjs',  
  defaultExtension: 'js',  
  main: 'index.js'  
}
```

```
'moment': {  
  main: 'moment.js'  
}
```

Examples

Adding jquery library in angular-cli project

1. Install jquery via npm :

```
npm install jquery --save
```

Install typings for the library:

To add typings for a library, do the following:

```
typings install jquery --global --save
```

2. Add jquery to angular-cli-build.js file to vendorNpmFiles array:

This is required so the build system will pick up the file. After setup the angular-cli-build.js should look like this:

Browse the `node_modules` and look for files and folders you want to add to the vendor folder.

```
var Angular2App = require('angular-cli/lib/broccoli/angular2-app');  
  
module.exports = function(defaults) {  
  return new Angular2App(defaults, {
```

```
vendorNpmFiles: [  
  // ...  
  'jquery/dist/*.js'  
  
  ]  
});  
};
```

3. Configure SystemJS mappings to know where to look for jquery :

SystemJS configuration is located in `system-config.ts` and after the custom configuration is done the related section should look like:

```
/** Map relative paths to URLs. */  
const map: any = {  
  'jquery': 'vendor/jquery'  
};  
  
/** User packages configuration. */  
const packages: any = {  
  
  // no need to add anything here for jquery  
  
};
```

4. In your `src/index.html` add this line

```
<script src="vendor/jquery/dist/jquery.min.js" type="text/javascript"></script>
```

Your other options are:

```
<script src="vendor/jquery/dist/jquery.js" type="text/javascript"></script>
```

or

```
<script src="/vendor/jquery/dist/jquery.slim.js" type="text/javascript"></script>
```

and

```
<script src="/vendor/jquery/dist/jquery.slim.min.js" type="text/javascript"></script>
```

5. Importing and using jquery library in your project source files:

Import jquery library in your source `.ts` files like this:

```
declare var $:any;  
  
@Component({
```

```

})
export class YourComponent {
  ngOnInit() {
    $(".button").click(function(){
      // now you can DO, what ever you want
    });
    console.log();
  }
}

```

If you followed the steps correctly you should now have jquery library working in your project. Enjoy!

Add 3rd party library that does not have typings

Notice, this is only for angular-cli up to 1.0.0-beta.10 version !

Some libraries or plugins may not have typings. Without these, TypeScript can't type check them and therefore causes compilation errors. These libraries can still be used but differently than imported modules.

1. Include a script reference to the library on your page (`index.html`)

```

<script src="//cdn.somewhe.re/lib.min.js" type="text/javascript"></script>
<script src="/local/path/to/lib.min.js" type="text/javascript"></script>

```

- These scripts should add a global (eg. `THREE`, `mapbox`, `$`, etc.) or attach to a global
2. In the component that requires these, use `declare` to initialize a variable matching the global name used by the lib. This lets TypeScript know that it has already been initialized. ¹

```
declare var <globalname>: any;
```

Some libs attach to `window`, which would need to be extended in order to be accessible in the app.

```
interface WindowIntercom extends Window { Intercom: any; }
declare var window: WindowIntercom;
```

3. Use the lib in your components as needed.

```

@Component { ... }
export class AppComponent implements AfterViewInit {
  ...
  ngAfterViewInit() {
    var geometry = new THREE.BoxGeometry( 1, 1, 1 );
    window.Intercom('boot', { ... }
  }
}

```

- NOTE: Some libs may interact with the DOM and should be used in the appropriate [component lifecycle](#) method.

Read [Installing 3rd party plugins with angular-cli@1.0.0-beta.10](https://riptutorial.com/angular2/topic/2328/installing-3rd-party-plugins-with-angular-cli-1-0-0-beta-10) online:

<https://riptutorial.com/angular2/topic/2328/installing-3rd-party-plugins-with-angular-cli-1-0-0-beta-10>

Chapter 50: Lazy loading a module

Examples

Lazy loading example

Lazy loading modules helps us decrease the startup time. With lazy loading our application does not need to load everything at once, it only needs to load what the user expects to see when the app first loads. Modules that are lazily loaded will only be loaded when the user navigates to their routes.

app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { EagerComponent } from './eager.component';
import { routing } from './app.routing';
@NgModule({
  imports: [
    BrowserModule,
    routing
  ],
  declarations: [
    AppComponent,
    EagerComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

app/app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `<h1>My App</h1>    <nav>
    <a routerLink="eager">Eager</a>
    <a routerLink="lazy">Lazy</a>
  </nav>
  <router-outlet></router-outlet>
`
})
export class AppComponent {}
```

app/app.routing.ts

```
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { EagerComponent } from './eager.component';
const routes: Routes = [
  { path: '', redirectTo: 'eager', pathMatch: 'full' },
```

```

    { path: 'eager', component: EagerComponent },
    { path: 'lazy', loadChildren: './lazy.module' }
  ];
  export const routing: ModuleWithProviders = RouterModule.forRoot(routes);

```

app/eager.component.ts

```

import { Component } from '@angular/core';
@Component({
  template: `

Eager Component</p>`
})
export class EagerComponent {}


```

There's nothing special about LazyModule other than it has its own routing and a component called LazyComponent (but it's not necessary to name your module or simliar so).

app/lazy.module.ts

```

import { NgModule } from '@angular/core';
import { LazyComponent } from './lazy.component';
import { routing } from './lazy.routing';
@NgModule({
  imports: [routing],
  declarations: [LazyComponent]
})
export class LazyModule {}

```

app/lazy.routing.ts

```

import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { LazyComponent } from './lazy.component';
const routes: Routes = [
  { path: '', component: LazyComponent }
];
export const routing: ModuleWithProviders = RouterModule.forChild(routes);

```

app/lazy.component.ts

```

import { Component } from '@angular/core';
@Component({
  template: `

Lazy Component</p>`
})
export class LazyComponent {}


```

Read Lazy loading a module online: <https://riptutorial.com/angular2/topic/7751/lazy-loading-a-module>

Chapter 51: Lifecycle Hooks

Remarks

Events availability

`AfterViewInit` and `AfterViewChecked` are only available in **Components**, and **not in Directives**.

Events order

- `OnChanges` (multiple times)
- `OnInit` (once)
- `DoCheck` (multiple times)
- `AfterContentInit` (once)
- `AfterContentChecked` (multiple times)
- `AfterViewInit` (once) (Component only)
- `AfterViewChecked` (multiple times) (Component only)
- `OnDestroy` (once)

Further Reading

- [Angular Documentation - Lifecycle Hooks](#)

Examples

OnInit

Fired when component or directive properties have been initialized.

(Before those of the child directives)

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'so-oninit-component',
  templateUrl: 'oninit-component.html',
  styleUrls: ['oninit-component.'],
})
class OnInitComponent implements OnInit {

  ngOnInit(): void {
    console.log('Component is ready !');
  }
}
```

OnDestroy

Fired when the component or directive instance is destroyed.

```
import { Component, OnDestroy } from '@angular/core';

@Component({
  selector: 'so-ondestroy-component',
  templateUrl: 'ondestroy-component.html',
  styleUrls: ['ondestroy-component.'],
})
class OnDestroyComponent implements OnDestroy {

  ngOnDestroy(): void {
    console.log('Component was destroyed !');
  }
}
```

OnChange

Fired when one or more of the component or directive properties have been changed.

```
import { Component, OnChanges, Input } from '@angular/core';

@Component({
  selector: 'so-onchanges-component',
  templateUrl: 'onchanges-component.html',
  styleUrls: ['onchanges-component.'],
})
class OnChangesComponent implements OnChanges {
  @Input() name: string;
  message: string;

  ngOnChanges(changes: SimpleChanges): void {
    console.log(changes);
  }
}
```

On change event will log

```
name: {
  currentValue: 'new name value',
  previousValue: 'old name value'
},
message: {
  currentValue: 'new message value',
  previousValue: 'old message value'
}
```

AfterContentInit

Fire after the initialization of the content of the component or directive has finished.

(Right after OnInit)

```
import { Component, AfterContentInit } from '@angular/core';

@Component({
  selector: 'so-aftercontentinit-component',
  templateUrl: 'aftercontentinit-component.html',
  styleUrls: ['aftercontentinit-component.Styles.css']
})
class AfterContentInitComponent implements AfterContentInit {

  ngAfterContentInit(): void {
    console.log('Component content have been loaded!');
  }
}
```

AfterContentChecked

Fire after the view has been fully initialized.

(Only available for components)

```
import { Component, AfterContentChecked } from '@angular/core';

@Component({
  selector: 'so-aftercontentchecked-component',
  templateUrl: 'aftercontentchecked-component.html',
  styleUrls: ['aftercontentchecked-component.Styles.css']
})
class AfterContentCheckedComponent implements AfterContentChecked {

  ngAfterContentChecked(): void {
    console.log('Component content have been checked!');
  }
}
```

AfterViewInit

Fires after initializing both the component view and any of its child views. This is a useful lifecycle hook for plugins outside of the Angular 2 ecosystem. For example, you could use this method to initialize a jQuery date picker based on the markup that Angular 2 has rendered.

```
import { Component, AfterViewInit } from '@angular/core';

@Component({
  selector: 'so-afterviewinit-component',
  templateUrl: 'afterviewinit-component.html',
  styleUrls: ['afterviewinit-component.Styles.css']
})
class AfterViewInitComponent implements AfterViewInit {

  ngAfterViewInit(): void {
    console.log('This event fire after the content init have been loaded!');
  }
}
```

AfterViewChecked

Fire after the check of the view, of the component, has finished.

(Only available for components)

```
import { Component, AfterViewChecked } from '@angular/core';

@Component({
  selector: 'so-afterviewchecked-component',
  templateUrl: 'afterviewchecked-component.html',
  styleUrls: ['afterviewchecked-component.']
})
class AfterViewCheckedComponent implements AfterViewChecked {

  ngAfterViewChecked(): void {
    console.log('This event fire after the content have been checked!');
  }
}
```

DoCheck

Allows to listen for changes only on specified properties

```
import { Component, DoCheck, Input } from '@angular/core';

@Component({
  selector: 'so-docheck-component',
  templateUrl: 'docheck-component.html',
  styleUrls: ['docheck-component.']
})
class DoCheckComponent implements DoCheck {
  @Input() elements: string[];
  differ: any;
  ngDoCheck(): void {
    // get value for elements property
    const changes = this.differ.diff(this.elements);

    if (changes) {
      changes.forEachAddedItem(res => console.log('Added', r.item));
      changes.forEachRemovedItem(r => console.log('Removed', r.item));
    }
  }
}
```

Read Lifecycle Hooks online: <https://riptutorial.com/angular2/topic/2935/lifecycle-hooks>

Chapter 52: Mocking @ngrx/Store

Introduction

@ngrx/Store is becoming more widely used in Angular 2 projects. As such, the Store is required to be injected into the constructor of any Component or Service that wishes to use it. Unit testing Store isn't as easy as testing a simple service though. As with many problems, there are a myriad of ways to implement solutions. However, the basic recipe is to write a mock class for the Observer interface and to write a mock class for Store. Then you can inject Store as a provider in your TestBed.

Parameters

name	description
value	next value to be observed
error	description
err	error to be thrown
super	description
action\$	mock Observer that does nothing unless defined to do so in the mock class
actionReducer\$	mock Observer that does nothing unless defined to do so in the mock class
obs\$	mock Observable

Remarks

Observer is a generic, but must be of type `any` to avoid unit testing complexity. The reason for this complexity, is that Store's constructor expects Observer arguments with different generic types. Using `any` avoids this complication.

It's possible to pass null values into StoreMock's super constructor, but this restricts the number of assertions that can be used to test the class further down the road.

The Component being used in this example is just being used as context for how one would go about injecting Store as a provide in the test setup.

Examples

Observer Mock

```

class ObserverMock implements Observer<any> {
  closed?: boolean = false; // inherited from Observer
  nextVal: any = ''; // variable I made up

  constructor() {}

  next = (value: any): void => { this.nextVal = value; };
  error = (err: any): void => { console.error(err); };
  complete = (): void => { this.closed = true; }
}

let actionReducer$: ObserverMock = new ObserverMock();
let action$: ObserverMock = new ObserverMock();
let obs$: Observable<any> = new Observable<any>();

class StoreMock extends Store<any> {
  constructor() {
    super(action$, actionReducer$, obs$);
  }
}

describe('Component:Typeahead', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [...],
      declarations: [Typeahead],
      providers: [
        {provide: Store, useClass: StoreMock} // NOTICE useClass instead of useValue
      ]
    }).compileComponents();
  });
});

```

Unit Test For Component With Mock Store

This is a unit test of a component that has *Store* as a dependency. Here, we are creating a new class called *MockStore* that is injected into our component instead of the usual *Store*.

```

import { Injectable } from '@angular/core';
import { TestBed, async } from '@angular/core/testing';
import { AppComponent } from './app.component';
import { DumbComponentComponent } from './dumb-component/dumb-component.component';
import { SmartComponentComponent } from './smart-component/smart-component.component';
import { mainReducer } from './state-management/reducers/main-reducer';
import { StoreModule } from '@ngrx/store';
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs';

class MockStore {
  public dispatch(obj) {
    console.log('dispatching from the mock store!')
  }

  public select(obj) {
    console.log('selecting from the mock store!');

    return Observable.of({})
  }
}

```



```

}

describe('AppComponent', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent,
        SmartComponentComponent,
        DumbComponentComponent,
      ],
      imports: [
        StoreModule.provideStore({mainReducer})
      ],
      providers: [
        {provide: Store, useClass: MockStore}
      ]
    });
  });

  it('should create the app', async(() => {

    let fixture = TestBed.createComponent(AppComponent);
    let app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();

  }));

```

Unit Test For Component Spying On Store

This is a unit test of a component that has *Store* as a dependency. Here, we are able to use a store with the default "initial state" while preventing it from actually dispatching actions when *store.dispatch()* is called.

```

import {TestBed, async} from '@angular/core/testing';
import {AppComponent} from './app.component';
import {DumbComponentComponent} from './dumb-component/dumb-component.component';
import {SmartComponentComponent} from './smart-component/smart-component.component';
import {mainReducer} from './state-management/reducers/main-reducer';
import {StoreModule} from '@ngrx/store';
import {Store} from '@ngrx/store';
import {Observable} from 'rxjs';

describe('AppComponent', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent,
        SmartComponentComponent,
        DumbComponentComponent,
      ],
      imports: [
        StoreModule.provideStore({mainReducer})
      ]
    });
  });

  });

```

```

it('should create the app', async(() => {
  let fixture = TestBed.createComponent(AppComponent);
  let app = fixture.debugElement.componentInstance;

  var mockStore = fixture.debugElement.injector.get(Store);
  var storeSpy = spyOn(mockStore, 'dispatch').and.callFake(function () {
    console.log('dispatching from the spy!');
  });

  });

});

```

Angular 2 - Mock Observable (service + component)

service

- I created post service with postRequest method.

```

import {Injectable} from '@angular/core';
import {Http, Headers, Response} from "@angular/http";
import {PostModel} from "./PostModel";
import 'rxjs/add/operator/map';
import {Observable} from "rxjs";

@Injectable()
export class PostService {

  constructor(private _http: Http) {
  }

  postRequest(postModel: PostModel) : Observable<Response> {
    let headers = new Headers();
    headers.append('Content-Type', 'application/json');
    return this._http.post("/postUrl", postModel, {headers})
      .map(res => res.json());
  }
}

```

Component

- I created component with result parameter and postExample function that call to postService.
- when the post request succeeded than result parameter should be 'Success' else 'Fail'

```

import {Component} from '@angular/core';
import {PostService} from "./PostService";
import {PostModel} from "./PostModel";

@Component({
  selector: 'app-post',
  templateUrl: './post.component.html',
  styleUrls: ['./post.component.scss'],
  providers : [PostService]
})

```

```

})
export class PostComponent{

  constructor(private _postService : PostService) {

    let postModel = new PostModel();
    result : string = null;
    postExample(){
      this._postService.postRequest(this.postModel)
        .subscribe(
          () => {
            this.result = 'Success';
          },
          err => this.result = 'Fail'
        )
    }
  }
}

```

test service

- when you want to test service that using http you should use mockBackend. and inject it to it.
- you need also to inject postService.

```

describe('Test PostService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpModule],
      providers: [
        PostService,
        MockBackend,
        BaseRequestOptions,
        {
          provide: Http,
          deps: [MockBackend, BaseRequestOptions],
          useFactory: (backend: XHRBackend, defaultOptions: BaseRequestOptions) => {
            return new Http(backend, defaultOptions);
          }
        }
      ]
    });
  });

  it('sendPostRequest function return Observable', inject([PostService, MockBackend],
(service: PostService, mockBackend: MockBackend) => {
    let mockPostModel = PostModel();

    mockBackend.connections.subscribe((connection: MockConnection) => {
      expect(connection.request.method).toEqual(RequestMethod.Post);
      expect(connection.request.url.indexOf('postUrl')).not.toEqual(-1);
      expect(connection.request.headers.get('Content-Type')).toEqual('application/json');
    });

    service
      .postRequest(PostModel)
      .subscribe((response) => {
        expect(response).toBeDefined();
      });
  });

```

```

    });
  });
});

```

test component

```

describe('testing post component', () => {
  let component: PostComponent;
  let fixture: ComponentFixture<postComponent>;

  let mockRouter = {
    navigate: jasmine.createSpy('navigate')
  };

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [PostComponent],
      imports: [RouterTestingModule.withRoutes([], ModalModule.forRoot() )],
      providers: [PostService, MockBackend, BaseRequestOptions,
        {provide: Http, deps: [MockBackend, BaseRequestOptions],
          useFactory: (backend: XHRBackend, defaultOptions: BaseRequestOptions) => {
            return new Http(backend, defaultOptions);
          }
        }
      ],
      {provide: Router, useValue: mockRouter}
    ],
    schemas: [ CUSTOM_ELEMENTS_SCHEMA ]
  }).compileComponents();
}));

beforeEach(() => {
  fixture = TestBed.createComponent(PostComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});

it('test postRequest success', inject([PostService, MockBackend], (service: PostService,
mockBackend: MockBackend) => {
  fixturePostComponent = TestBed.createComponent(PostComponent);
  componentPostComponent = fixturePostComponent.componentInstance;
  fixturePostComponent.detectChanges();

  component.postExample();
  let postModel = new PostModel();
  let response = {
    'message' : 'message',
    'ok'      : true
  };
  mockBackend.connections.subscribe((connection: MockConnection) => {
    postComponent.result = 'Success'
    connection.mockRespond(new Response(
      new ResponseOptions({
        body: response
      })
    ))
  })
}));

```

```

    });
    service.postRequest(postModel)
      .subscribe((data) => {
        expect(component.result).toBeDefined();
        expect(PostComponent.result).toEqual('Success');
        expect(data).toEqual(response);
      });
  });
});
});

```

Simple Store

simple.action.ts

```

import { Action } from '@ngrx/store';

export enum simpleActionType {
  add = "simpleAction_Add",
  add_Success = "simpleAction_Add_Success"
}

export class simpleAction {
  type: simpleActionType
  constructor(public payload: number) { }
}

```

simple.effects.ts

```

import { Effect, Actions } from '@ngrx/effects';
import { Injectable } from '@angular/core';
import { Action } from '@ngrx/store';
import { Observable } from 'rxjs';

import { simpleAction, simpleActionType } from './simple.action';

@Injectable()
export class simpleEffects {

  @Effect()
  addAction$: Observable<simpleAction> = this.actions$
    .ofType(simpleActionType.add)
    .switchMap((action: simpleAction) => {
      console.log(action);

      return Observable.of({ type: simpleActionType.add_Success, payload: action.payload
    });

    // if you have an api use this code
    // return this.http.post(url).catch().map(res=>{ type: simpleAction.add_Success,
    payload:res})
  });
  constructor(private actions$: Actions) { }
}

```

simple.reducer.ts

```

import { Action, ActionReducer } from '@ngrx/store';

import { simpleAction, simpleActionType } from './simple.action';

export const simpleReducer: ActionReducer<number> = (state: number = 0, action: simpleAction):
number => {
  switch (action.type) {
    case simpleActionType.add_Success:
      console.log(action);
      return state + action.payload;
    default:
      return state;
  }
}

```

store/index.ts

```

import { combineReducers, ActionReducer, Action, StoreModule } from '@ngrx/store';
import { EffectsModule } from '@ngrx/effects';
import { ModuleWithProviders } from '@angular/core';
import { compose } from '@ngrx/core';

import { simpleReducer } from "../simple/simple.reducer";
import { simpleEffects } from "../simple/simple.effects";

export interface IAppState {
  sum: number;
}

// all new reducers should be define here
const reducers = {
  sum: simpleReducer
};

export const store: ModuleWithProviders = StoreModule.forRoot(reducers);
export const effects: ModuleWithProviders[] = [
  EffectsModule.forRoot([simpleEffects])
];

```

app.module.ts

```

import { BrowserModule } from '@angular/platform-browser'
import { NgModule } from '@angular/core';

import { effects, store } from "../Store/index";
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    // store
    store,
    effects
  ],
  providers: [],

```

```
bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.ts

```
import { Component } from '@angular/core';

import { Store } from '@ngrx/store';
import { Observable } from 'rxjs';

import { IAppState } from './Store/index';
import { simpleActionTpye } from './Store/simple/simple.action';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';

  constructor(private store: Store<IAppState>) {
    store.select(s => s.sum).subscribe((res) => {
      console.log(res);
    })
    this.store.dispatch({
      type: simpleActionTpye.add,
      payload: 1
    })
    this.store.dispatch({
      type: simpleActionTpye.add,
      payload: 2
    })
    this.store.dispatch({
      type: simpleActionTpye.add,
      payload: 3
    })
  }
}
```

result 0 1 3 6

Read Mocking @ngrx/Store online: <https://riptutorial.com/angular2/topic/8038/mocking--ngrx-store>

Chapter 53: Modules

Introduction

Angular modules are containers for different parts of your app.

You can have nested modules, your `app.module` is already actually nesting other modules such as `BrowserModule` and you can add `RouterModule` and so on.

Examples

A simple module

A module is a class with the `@NgModule` decorator. To create a module we add `@NgModule` passing some parameters:

- `bootstrap`: The component that will be the root of your application. This configuration is only present on your root module
- `declarations`: Resources the module declares. When you add a new component you have to update the declarations (`ng generate component` does it automatically)
- `exports`: Resources the module exports that can be used in other modules
- `imports`: Resources the module uses from other modules (only module classes are accepted)
- `providers`: Resources that can be injected (di) in a component

A simple example:

```
import { AppComponent } from './app.component';
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

@NgModule({
  bootstrap: [AppComponent]
  declarations: [AppComponent],
  exports: [],
  imports: [BrowserModule],
  providers: [],
})
export class AppModule { }
```

Nesting modules

Modules can be nested by using the `imports` parameter of `@NgModule` decorator.

We can create a `core.module` in our application that will contain generic things, like a `ReversePipe` (a pipe that reverse a string) and bundle those in this module:

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';
```



```
import { ReversePipe } from '../reverse.pipe';

@NgModule({
  imports: [
    CommonModule
  ],
  exports: [ReversePipe], // export things to be imported in another module
  declarations: [ReversePipe],
})
export class CoreModule { }
```

Then in the `app.module`:

```
import { CoreModule } from 'app/core/core.module';

@NgModule({
  declarations: [...], // ReversePipe is available without declaring here
                        // because CoreModule exports it
  imports: [
    CoreModule,        // import things from CoreModule
    ...
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Read Modules online: <https://riptutorial.com/angular2/topic/10840/modules>

Chapter 54: ngrx

Introduction

Ngrx is a powerful library that you can use with **Angular2**. The idea behind is to merge two concepts that plays well together to have a **reactive app** with a predictable **state container** : - [Redux][1] - [RxJs][2] The main advantages : - Sharing data in your app between your components is going to easier - Testing your app core logic consists to test pure functions, without any dependency on Angular2 (very easy so !) [1]: <http://redux.js.org> [2]: <http://reactivex.io/rxjs>

Examples

Complete example : Login/logout a user

Prerequisites

This topic is **not** about Redux and/or Ngrx :

- You need to be comfortable with Redux
- At least understand the basics of RxJs and Observable pattern

First, let's define an example from the very beginning and play with some code :

As a developer, I want to :

1. Have an `IUser` interface that defines the properties of a `User`
2. Declare the actions that we'll use later to manipulate the `User` in the `Store`
3. Define the initial state of the `UserReducer`
4. Create the reducer `UserReducer`
5. Import our `UserReducer` into our main module to build the `Store`
6. Use data from the `Store` to display information in our view

Spoiler alert : If you want to try the demo right away or read the code before we even get started, here's a Plunkr ([embed view](#) or [run view](#)).

1) Define `IUser` interface

I like to split my interfaces in two parts :

- The properties we'll get from a server
- The properties we define only for the UI (should a button be spinning for example)

And here's the interface `IUser` we'll be using :

```
user.interface.ts
```

```
export interface IUser {
  // from server
  username: string;
  email: string;

  // for UI
  isConnected: boolean;
  isConnecting: boolean;
};
```

2) Declare the actions to manipulate the User

Now we've got to think about what kind of actions our *reducers* are supposed to handle. Let say here :

user.actions.ts

```
export const UserActions = {
  // when the user clicks on login button, before we launch the HTTP request
  // this will allow us to disable the login button during the request
  USR_IS_CONNECTING: 'USR_IS_CONNECTING',
  // this allows us to save the username and email of the user
  // we assume those data were fetched in the previous request
  USR_IS_CONNECTED: 'USR_IS_CONNECTED',

  // same pattern for disconnecting the user
  USR_IS_DISCONNECTING: 'USR_IS_DISCONNECTING',
  USR_IS_DISCONNECTED: 'USR_IS_DISCONNECTED'
};
```

But before we use those actions, let me explain why we're going to need a service to dispatch **some** of those actions for us :

Let say that we want to connect a user. So we'll be clicking on a login button and here's what's going to happen :

- Click on the button
- The component catch the event and call `userService.login`
- `userService.login` method dispatch an event to update our store property : `user.isConnected`
- An HTTP call is fired (we'll use a `setTimeout` in the demo to simulate the **async behaviour**)
- Once the HTTP call is finished, we'll dispatch another action to warn our store that a user is logged

user.service.ts

```
@Injectable()
export class UserService {
  constructor(public store$: Store<AppState>) { }

  login(username: string) {
    // first, dispatch an action saying that the user's trying to connect
    // so we can lock the button until the HTTP request finish
  }
}
```

```

this.store$.dispatch({ type: UserActions.USR_IS_CONNECTING });

// simulate some delay like we would have with an HTTP request
// by using a timeout
setTimeout(() => {
  // some email (or data) that you'd have get as HTTP response
  let email = `${username}@email.com`;

  this.store$.dispatch({ type: UserActions.USR_IS_CONNECTED, payload: { username, email }
});
}, 2000);
}

logout() {
  // first, dispatch an action saying that the user's trying to connect
  // so we can lock the button until the HTTP request finish
  this.store$.dispatch({ type: UserActions.USR_IS_DISCONNECTING });

  // simulate some delay like we would have with an HTTP request
  // by using a timeout
  setTimeout(() => {
    this.store$.dispatch({ type: UserActions.USR_IS_DISCONNECTED });
  }, 2000);
}
}

```

3) Define the initial state of the `UserReducer`

user.state.ts

```

export const UserFactory: IUser = () => {
  return {
    // from server
    username: null,
    email: null,

    // for UI
    isConnecting: false,
    isConnected: false,
    isDisconnecting: false
  };
};

```

4) Create the reducer `UserReducer`

A reducer takes 2 arguments :

- The current state
- An `Action` of type `Action<{type: string, payload: any}>`

Reminder : A reducer needs to be initialized at some point

As we defined the default state of our reducer in part 3), we'll be able to use it like that :

user.reducer.ts

```
export const UserReducer: ActionReducer<IUser> = (user: IUser, action: Action) => {
  if (user === null) {
    return userFactory();
  }

  // ...
}
```

Hopefully, there's an easier way to write that by using our `factory` function to return an object and within the reducer use an (ES6) [default parameters value](#) :

```
export const UserReducer: ActionReducer<IUser> = (user: IUser = UserFactory(), action: Action)
=> {
  // ...
}
```

Then, we need to handle every actions in our reducer : **TIP**: Use [ES6 Object.assign](#) function to keep our state immutable

```
export const UserReducer: ActionReducer<IUser> = (user: IUser = UserFactory(), action: Action)
=> {
  switch (action.type) {
    case UserActions.USR_IS_CONNECTING:
      return Object.assign({}, user, { isConnecting: true });

    case UserActions.USR_IS_CONNECTED:
      return Object.assign({}, user, { isConnecting: false, isConnected: true, username:
action.payload.username });

    case UserActions.USR_IS_DISCONNECTING:
      return Object.assign({}, user, { isDisconnecting: true });

    case UserActions.USR_IS_DISCONNECTED:
      return Object.assign({}, user, { isDisconnecting: false, isConnected: false });

    default:
      return user;
  }
};
```

5) Import our `UserReducer` into our main module to build the `Store`

app.module.ts

```
@NgModule({
  declarations: [
    AppComponent
```

```

],
imports: [
// angular modules
// ...

// declare your store by providing your reducers
// (every reducer should return a default state)
StoreModule.provideStore({
  user: UserReducer,
  // of course, you can put as many reducers here as you want
  // ...
}),

// other modules to import
// ...
]
});

```

6) Use data from the `Store` to display information in our view

Everything is now ready on logic side and we just have to display what we want in two components :

- **UserComponent: [Dumb component]** We'll just pass the user object from the store using `@Input` property and `async` pipe. This way, the component will receive the user only once it's available (and the `user` will be of type `IUser` and not of type `Observable<IUser>` !)
- **LoginComponent [Smart component]** We'll directly inject the `Store` into this component and work only on `user` as an `Observable`.

user.component.ts

```

@Component({
  selector: 'user',
  styles: [
    '.table { max-width: 250px; }',
    '.truthy { color: green; font-weight: bold; }',
    '.falsy { color: red; }'
  ],
  template: `
    <h2>User information :</h2>

    <table class="table">
      <tr>
        <th>Property</th>
        <th>Value</th>
      </tr>

      <tr>
        <td>username</td>
        <td [class.truthy]="user.username" [class.falsy]="!user.username">
          {{ user.username ? user.username : 'null' }}
        </td>
      </tr>
    </table>
  `
})

```

```

<tr>
  <td>email</td>
  <td [class.truthy]="user.email" [class.falsy]="!user.email">
    {{ user.email ? user.email : 'null' }}
  </td>
</tr>

<tr>
  <td>isConnecting</td>
  <td [class.truthy]="user.isConnecting" [class.falsy]="!user.isConnecting">
    {{ user.isConnecting }}
  </td>
</tr>

<tr>
  <td>isConnected</td>
  <td [class.truthy]="user.isConnected" [class.falsy]="!user.isConnected">
    {{ user.isConnected }}
  </td>
</tr>

<tr>
  <td>isDisconnecting</td>
  <td [class.truthy]="user.isDisconnecting" [class.falsy]="!user.isDisconnecting">
    {{ user.isDisconnecting }}
  </td>
</tr>
</table>
,
})
export class UserComponent {
  @Input() user;

  constructor() { }
}

```

login.component.ts

```

@Component({
  selector: 'login',
  template: `
    <form
      *ngIf="!(user | async).isConnected"
      #loginForm="ngForm"
      (ngSubmit)="login(loginForm.value.username)"
    >
      <input
        type="text"
        name="username"
        placeholder="Username"
        [disabled]="(user | async).isConnecting"
        ngModel
      >

      <button
        type="submit"
        [disabled]="(user | async).isConnecting || (user | async).isConnected"
      >Log me in</button>
    </form>

```

```

    <button
      *ngIf="(user | async).isConnected"
      (click)="logout()"
      [disabled]="(user | async).isDisconnecting"
    >Log me out</button>
  `
})
export class LoginComponent {
  public user: Observable<IUser>;

  constructor(public store$: Store<AppState>, private userService: UserService) {
    this.user = store$.select('user');
  }

  login(username: string) {
    this.userService.login(username);
  }

  logout() {
    this.userService.logout();
  }
}

```

As `Ngrx` is a merge of `Redux` and `RxJs` concepts, it can be quite hard to understand the ins and outs at the beginning. But this is a powerful pattern that allows you as we've seen in this example to have a *reactive app* and where you can easily share your data. Don't forget that there's a [Plunker](#) available and you can fork it to make your own tests !

I hope it was helpful even tho the topic is quite long, cheers !

Read `ngrx` online: <https://riptutorial.com/angular2/topic/8086/ngrx>

Chapter 55: Optimizing rendering using ChangeDetectionStrategy

Examples

Default vs OnPush

Consider the following component with one input `myInput` and an internal value called `someInternalValue`. Both of them are used in a component's template.

```
import {Component, Input} from '@angular/core';

@Component ({
  template:`
  <div>
    <p>{{myInput}}</p>
    <p>{{someInternalValue}}</p>
  </div>
  `
})
class MyComponent {
  @Input () myInput: any;

  someInternalValue: any;

  // ...
}
```

By default, the `changeDetection:` property in the component decorator will be set to `ChangeDetectionStrategy.Default`; implicit in the example. In this situation, any changes to any of the values in the template will trigger a re-render of `MyComponent`. In other words, if I change `myInput` or `someInternalValue` angular 2 will exert energy and re-render the component.

Suppose, however, that we only want to re-render when the inputs change. Consider the following component with `changeDetection:` set to `ChangeDetectionStrategy.OnPush`

```
import {Component, ChangeDetectionStrategy, Input} from '@angular/core';

@Component ({
  changeDetection: ChangeDetectionStrategy.OnPush
  template:`
  <div>
    <p>{{myInput}}</p>
    <p>{{someInternalValue}}</p>
  </div>
  `
})
class MyComponent {
  @Input () myInput: any;

  someInternalValue: any;
```

```
// ...  
}
```

By setting `changeDetection: to ChangeDetectionStrategy.OnPush`, `MyComponent` will only re-render when its inputs change. In this case, `myInput` will need to receive a new value from its parent to trigger a re-render.

Read [Optimizing rendering using ChangeDetectionStrategy](https://riptutorial.com/angular2/topic/2644/optimizing-rendering-using-changedetectionstrategy) online:

<https://riptutorial.com/angular2/topic/2644/optimizing-rendering-using-changedetectionstrategy>

Chapter 56: OrderBy Pipe

Introduction

How to write order pipe and use it.

Examples

The Pipe

The Pipe implementation

```
import {Pipe, PipeTransform} from '@angular/core';

@Pipe({
  name: 'orderBy',
  pure: false
})
export class OrderBy implements PipeTransform {

  value:string[] =[];

  static _orderByComparator(a:any, b:any):number{

    if(a === null || typeof a === 'undefined') a = 0;
    if(b === null || typeof b === 'undefined') b = 0;

    if((isNaN(parseFloat(a)) || !isFinite(a)) || (isNaN(parseFloat(b)) || !isFinite(b))){
      //Isn't a number so lowercase the string to properly compare
      if(a.toLowerCase() < b.toLowerCase()) return -1;
      if(a.toLowerCase() > b.toLowerCase()) return 1;
    }else{
      //Parse strings as numbers to compare properly
      if(parseFloat(a) < parseFloat(b)) return -1;
      if(parseFloat(a) > parseFloat(b)) return 1;
    }

    return 0; //equal each other
  }

  transform(input:any, config:string = '+'): any{

    //make a copy of the input's reference
    this.value = [...input];
    let value = this.value;

    if(!Array.isArray(value)) return value;

    if(!Array.isArray(config) || (Array.isArray(config) && config.length === 1)){
      let propertyToCheck:string = !Array.isArray(config) ? config : config[0];
      let desc = propertyToCheck.substr(0, 1) === '-';

      //Basic array
      if(!propertyToCheck || propertyToCheck === '-' || propertyToCheck === '+'){
```

```

    return !desc ? value.sort() : value.sort().reverse();
  } else {
    let property:string = propertyToCheck.substr(0, 1) === '+' ||
propertyToCheck.substr(0, 1) === '-'
      ? propertyToCheck.substr(1)
      : propertyToCheck;

    return value.sort(function(a:any,b:any){
      return !desc
        ? OrderBy._orderByComparator(a[property], b[property])
        : -OrderBy._orderByComparator(a[property], b[property]);
    });
  }
} else {
  //Loop over property of the array in order and sort
  return value.sort(function(a:any,b:any){
    for(let i:number = 0; i < config.length; i++){
      let desc = config[i].substr(0, 1) === '-';
      let property = config[i].substr(0, 1) === '+' || config[i].substr(0, 1) === '-'
        ? config[i].substr(1)
        : config[i];

      let comparison = !desc
        ? OrderBy._orderByComparator(a[property], b[property])
        : -OrderBy._orderByComparator(a[property], b[property]);

      //Don't return 0 yet in case of needing to sort by next property
      if(comparison !== 0) return comparison;
    }

    return 0; //equal each other
  });
}
}
}
}

```

How to use the pipe in the HTML - order ascending by first name

```

<table>
  <thead>
    <tr>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Age</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let user of users | orderBy : ['firstName']">
      <td>{{user.firstName}}</td>
      <td>{{user.lastName}}</td>
      <td>{{user.age}}</td>
    </tr>
  </tbody>
</table>

```

How to use the pipe in the HTML - order descending by first name

```
<table>
  <thead>
    <tr>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Age</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let user of users | orderBy : ['-firstName']">
      <td>{{user.firstName}}</td>
      <td>{{user.lastName}}</td>
      <td>{{user.age}}</td>
    </tr>
  </tbody>
</table>
```

Read OrderBy Pipe online: <https://riptutorial.com/angular2/topic/8969/orderby-pipe>

Chapter 57: Page title

Introduction

How can you change the title of the page

Syntax

- setTitle(newTitle: string): void;
- getTitle(): string;

Examples

changing the page title

1. First we need to provide Title service.
2. Using setTitle

```
import {Title} from "@angular/platform-browser";
@Component({
  selector: 'app',
  templateUrl: './app.component.html',
  providers : [Title]
})

export class AppComponent implements {
  constructor( private title: Title) {
    this.title.setTitle('page title changed');
  }
}
```

Read Page title online: <https://riptutorial.com/angular2/topic/8954/page-title>

Chapter 58: Pipes

Introduction

The pipe `|` character is used to apply pipes in Angular 2. Pipes are very similar to filters in AngularJS in that they both help to transform the data into a specified format.

Parameters

Function/Parameter	Explanation
<code>@Pipe({name, pure})</code>	metadata for pipe, must immediately precede pipe class
name: <i>string</i>	what you will use inside the template
pure: <i>boolean</i>	defaults to true, mark this as false to have your pipe re-evaluated more often
<code>transform(value, args[]?)</code>	the function that is called to transform the values in the template
value: <i>any</i>	the value that you want to transform
args: <i>any[]</i>	the arguments that you may need included in your transform. Mark optional args with the <code>?</code> operator like so <code>transform(value, arg1, arg2?)</code>

Remarks

This topic covers [Angular2 Pipes](#), a mechanism for transforming and formatting data within HTML templates in an Angular2 application.

Examples

Chaining Pipes

Pipes may be chained.

```
<p>Today is {{ today | date:'fullDate' | uppercase}}.</p>
```

Custom Pipes

my.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'myPipe'})
export class MyPipe implements PipeTransform {

  transform(value:any, args?: any):string {
    let transformedValue = value; // implement your transformation logic here
    return transformedValue;
  }
}
```

my.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-component',
  template: `{{ value | myPipe }}`
})
export class MyComponent {

  public value:any;
}
```

my.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { MyComponent } from './my.component';
import { MyPipe } from './my.pipe';

@NgModule({
  imports: [
    BrowserModule,
  ],
  declarations: [
    MyComponent,
    MyPipe
  ],
})
export class MyModule { }
```

Built-in Pipes

Angular2 comes with a few built-in pipes:

Pipe	Usage	Example
DatePipe	date	{{ dateObj date }} // output is 'Jun 15, 2015'
UpperCasePipe	uppercase	{{ value uppercase }} // output is 'SOMETEXT'
LowerCasePipe	lowercase	{{ value lowercase }} // output is 'sometext'

Pipe	Usage	Example
CurrencyPipe	currency	{{ 31.00 currency:'USD':true }} // output is '\$31'
PercentPipe	percent	{{ 0.03 percent }} //output is %3

There are others. Look [here](#) for their documentation.

Example

hotel-reservation.component.ts

```
import { Component } from '@angular/core';

@Component({
  moduleId: module.id,
  selector: 'hotel-reservation',
  templateUrl: './hotel-reservation.template.html'
})
export class HotelReservationComponent {
  public fName: string = 'Joe';
  public lName: string = 'SCHMO';
  public reservationMade: string = '2016-06-22T07:18-08:00'
  public reservationFor: string = '2025-11-14';
  public cost: number = 99.99;
}
```

hotel-reservation.template.html

```
<div>
  <h1>Welcome back {{fName | uppercase}} {{lName | lowercase}}</h1>
  <p>
    On {reservationMade | date} at {reservationMade | date:'shortTime'} you
    reserved room 205 for {reservationDate | date} for a total cost of
    {cost | currency}.
  </p>
</div>
```

Output

```
Welcome back JOE schmo
On Jun 26, 2016 at 7:18 you reserved room 205 for Nov 14, 2025 for a total cost of
$99.99.
```

Debugging With JsonPipe

The JsonPipe can be used for debugging the state of any given internal.

Code

```

@Component({
  selector: 'json-example',
  template: `<div>
    <p>Without JSON pipe:</p>
    <pre>{{object}}</pre>
    <p>With JSON pipe:</p>
    <pre>{{object | json}}</pre>
  </div>`
})
export class JsonPipeExample {
  object: Object = {foo: 'bar', baz: 'qux', nested: {xyz: 3, numbers: [1, 2, 3, 4, 5]}};
}

```

Output

```

Without JSON Pipe:
object
With JSON pipe:
{object:{foo: 'bar', baz: 'qux', nested: {xyz: 3, numbers: [1, 2, 3, 4, 5]}}

```

Globally Available Custom Pipe

To make a custom pipe available application wide, During application bootstrap, extending PLATFORM_PIPES.

```

import { bootstrap } from '@angular/platform-browser-dynamic';
import { provide, PLATFORM_PIPES } from '@angular/core';

import { AppComponent } from './app.component';
import { MyPipe } from './my.pipe'; // your custom pipe

bootstrap(AppComponent, [
  provide(PLATFORM_PIPES, {
    useValue: [
      MyPipe
    ],
    multi: true
  })
]);

```

Tutorial here: <https://scotch.io/tutorials/create-a-globally-available-custom-pipe-in-angular-2>

Creating Custom Pipe

app/pipes.pipe.ts

```

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'truthy'})
export class Truthy implements PipeTransform {
  transform(value: any, truthy: string, falsey: string): any {
    if (typeof value === 'boolean'){return value ? truthy : falsey;}
    else return value
  }
}

```

```
}
```

app/my-component.component.ts

```
import { Truthy } from './pipes.pipe';

@Component({
  selector: 'my-component',
  template: `
    <p>{{value | truthy:'enabled':'disabled'}}</p>
  `,
  pipes: [Truthy]
})
export class MyComponent{ }
```

Unwrap async values with async pipe

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/of';

@Component({
  selector: 'async-stuff',
  template: `
    <h1>Hello, {{ name | async }}</h1>
    Your Friends are:
    <ul>
      <li *ngFor="let friend of friends | async">
        {{friend}}
      </li>
    </ul>
  `,
})
class AsyncStuffComponent {
  name = Promise.resolve('Misko');
  friends = Observable.of(['Igor']);
}
```

Becomes:

```
<h1>Hello, Misko</h1>
Your Friends are:
<ul>
  <li>
    Igor
  </li>
</ul>
```

Extending an Existing Pipe

```
import { Pipe, PipeTransform } from '@angular/core';
import { DatePipe } from '@angular/common'

@Pipe({name: 'ifDate'})
```

```

export class IfDate implements PipeTransform {
  private datePipe: DatePipe = new DatePipe();

  transform(value: any, pattern?:string) : any {
    if (typeof value === 'number') {return value}
    try {
      return this.datePipe.transform(value, pattern)
    } catch(err) {
      return value
    }
  }
}

```

Stateful Pipes

Angular 2 offers two different types of pipes - stateless and stateful. Pipes are stateless by default. However, we can implement stateful pipes by setting the `pure` property to `false`. As you can see in the parameter section, you can specify a `name` and declare whether the pipe should be pure or not, meaning stateful or stateless. While data flows through a stateless pipe (which is a pure function) that **does not** remember anything, data can be managed and remembered by stateful pipes. A good example of a stateful pipe is the `AsyncPipe` that is provided by Angular 2.

Important

Notice that most pipes should fall into the category of stateless pipes. That's important for performance reasons since Angular can optimize stateless pipes for the change detector. So use stateful pipes cautiously. In general, the optimization of pipes in Angular 2 have a major performance enhancement over filters in Angular 1.x. In Angular 1 the digest cycle always had to re-run all filters even though the data hasn't changed at all. In Angular 2, once a pipe's value has been computed, the change detector knows not to run this pipe again unless the input changes.

Implementation of a stateful pipe

```

import {Pipe, PipeTransform, OnDestroy} from '@angular/core';

@Pipe({
  name: 'countdown',
  pure: false
})
export class CountdownPipe implements PipeTransform, OnDestroy {
  private interval: any;
  private remainingTime: number;

  transform(value: number, interval: number = 1000): number {
    if (!parseInt(value, 10)) {
      return null;
    }

    if (typeof this.remainingTime !== 'number') {
      this.remainingTime = parseInt(value, 10);
    }

    if (!this.interval) {
      this.interval = setInterval(() => {
        this.remainingTime--;
      }, interval);
    }
  }
}

```

```

        if (this.remainingTime <= 0) {
            this.remainingTime = 0;
            clearInterval(this.interval);
            delete this.interval;
        }
    }, interval);
}

return this.remainingTime;
}

ngOnDestroy(): void {
    if (this.interval) {
        clearInterval(this.interval);
    }
}
}
}

```

You can then use the pipe as usual:

```

{{ 1000 | countdown:50 }}
{{ 300 | countdown }}

```

It's important that your pipe also implements the `OnDestroy` interface so you can clean up once your pipe gets destroyed. In the example above, it's necessary to clear the interval to avoid memory leaks.

Dynamic Pipe

Use case scenario: A table view consists of different columns with different data format that needs to be transformed with different pipes.

table.component.ts

```

...
import { DYNAMIC_PIPES } from '../pipes/dynamic.pipe.ts';

@Component({
    ...
    pipes: [DYNAMIC_PIPES]
})
export class TableComponent {
    ...

    // pipes to be used for each column
    table.pipes = [ null, null, null, 'humanizeDate', 'statusFromBoolean' ],
    table.header = [ 'id', 'title', 'url', 'created', 'status' ],
    table.rows = [
        [ 1, 'Home', 'home', '2016-08-27T17:48:32', true ],
        [ 2, 'About Us', 'about', '2016-08-28T08:42:09', true ],
        [ 4, 'Contact Us', 'contact', '2016-08-28T13:28:18', false ],
        ...
    ]
    ...
}

```

```
}
```

dynamic.pipe.ts

```
import {
  Pipe,
  PipeTransform
} from '@angular/core';
// Library used to humanize a date in this example
import * as moment from 'moment';

@Pipe({name: 'dynamic'})
export class DynamicPipe implements PipeTransform {

  transform(value:string, modifier:string) {
    if (!modifier) return value;
    // Evaluate pipe string
    return eval('this.' + modifier + '(\'' + value + '\')')
  }

  // Returns 'enabled' or 'disabled' based on input value
  statusFromBoolean(value:string):string {
    switch (value) {
      case 'true':
      case '1':
        return 'enabled';
      default:
        return 'disabled';
    }
  }

  // Returns a human friendly time format e.g: '14 minutes ago', 'yesterday'
  humanizeDate(value:string):string {
    // Humanize if date difference is within a week from now else returns 'December 20,
    2016' format
    if (moment().diff(moment(value), 'days') < 8) return moment(value).fromNow();
    return moment(value).format('MMMM Do YYYY');
  }
}

export const DYNAMIC_PIPES = [DynamicPipe];
```

table.component.html

```
<table>
  <thead>
    <td *ngFor="let head of data.header">{{ head }}</td>
  </thead>
  <tr *ngFor="let row of table.rows; let i = index">
    <td *ngFor="let column of row">{{ column | dynamic:table.pipes[i] }}</td>
  </tr>
</table>
```

Result

ID	Page Title	Page URL	Created	Status
----	------------	----------	---------	--------

	1	Home	home	4 minutes ago	Enabled	
	2	About Us	about	Yesterday	Enabled	
	4	Contact Us	contact	Yesterday	Disabled	

Testing a pipe

Given a pipe that reverse a string

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'reverse' })
export class ReversePipe implements PipeTransform {
  transform(value: string): string {
    return value.split('').reverse().join('');
  }
}
```

It can be tested configuring the spec file like this

```
import { TestBed, inject } from '@angular/core/testing';

import { ReversePipe } from './reverse.pipe';

describe('ReversePipe', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [ReversePipe],
    });
  });

  it('should be created', inject([ReversePipe], (reversePipe: ReversePipe) => {
    expect(reversePipe).toBeTruthy();
  }));

  it('should reverse a string', inject([ReversePipe], (reversePipe: ReversePipe) => {
    expect(reversePipe.transform('abc')).toEqual('cba');
  }));
});
```

Read Pipes online: <https://riptutorial.com/angular2/topic/1165/pipes>

Chapter 59: Routing

Examples

Basic Routing

Router enables navigation from one view to another based on user interactions with the application.

Following are the steps in implementing basic routing in Angular 2 -

Basic precaution: Ensure you have the tag

```
<base href='/>
```

as the first child under your head tag in your index.html file. This tag tells that your app folder is the application root. Angular 2 would then know to organize your links.

First step is to check if you are pointing to correct/latest routing dependencies in package.json -

```
"dependencies": {  
  .....  
  "@angular/router": "3.0.0-beta.1",  
  .....  
}
```

Second step is to define the route as per it's class definition -

```
class Route {  
  path : string  
  pathMatch : 'full'|'prefix'  
  component : Type|string  
  .....  
}
```

In a routes file (`route/routes.ts`), import all the components which you need to configure for different routing paths. Empty path means that view is loaded by default. ":" in the path indicates dynamic parameter passed to the loaded component.

Routes are made available to application via dependency injection. `ProviderRouter` method is called with `RouterConfig` as parameter so that it can be injected to the components for calling routing specific tasks.

```
import { provideRouter, RouterConfig } from '@angular/router';  
import { BarDetailComponent } from '../components/bar-detail.component';  
import { DashboardComponent } from '../components/dashboard.component';  
import { LoginComponent } from '../components/login.component';  
import { SignupComponent } from '../components/signup.component';
```



```

export const appRoutes: RouterConfig = [
  { path: '', pathMatch: 'full', redirectTo: 'login' },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'bars/:id', component: BarDetailComponent },
  { path: 'login', component: LoginComponent },
  { path: 'signup', component: SignupComponent }
];

export const APP_ROUTER_PROVIDER = [provideRouter(appRoutes)];

```

Third step is to bootstrap the route provider.

In your `main.ts` (It can be any name. basically, it should your main file defined in `systemjs.config`)

```

import { bootstrap } from '@angular/platform-browser-dynamic';
import { AppComponent } from './components/app.component';
import { APP_ROUTER_PROVIDER } from './routes/routes';

bootstrap(AppComponent, [ APP_ROUTER_PROVIDER ]).catch(err => console.error(err));

```

Fourth step is to load/display the router components based on path accessed. directive is used to tell angular where to load the component. To use import the `ROUTER_DIRECTIVES`.

```

import { ROUTER_DIRECTIVES } from '@angular/router';

@Component({
  selector: 'demo-app',
  template: `
    .....
    <div>
      <router-outlet></router-outlet>
    </div>
    .....
  `,
  // Add our router directives we will be using
  directives: [ROUTER_DIRECTIVES]
})

```

Fifth step is to link the other routes. By default, RouterOutlet will load the component for which empty path is specified in the RouterConfig. RouterLink directive is used with html anchor tag to load the components attached to routes. RouterLink generates the href attribute which is used to generate links. For Ex:

```

import { Component } from '@angular/core';
import { ROUTER_DIRECTIVES } from '@angular/router';

@Component({
  selector: 'demo-app',
  template: `
    <a [routerLink]="['/login']">Login</a>
    <a [routerLink]="['/signup']">Signup</a>
    <a [routerLink]="['/dashboard']">Dashboard</a>
    <div>
      <router-outlet></router-outlet>
    </div>
  `,

```

```

// Add our router directives we will be using
directives: [ROUTER_DIRECTIVES]
})
export class AppComponent { }

```

Now, we are good with routing to static path. RouterLink support dynamic path also by passing extra parameters along with the path.

```

import { Component } from '@angular/core'; import { ROUTER_DIRECTIVES } from '@angular/router';

```

```

@Component({
  selector: 'demo-app',
  template: `
    <ul>
      <li *ngFor="let bar of bars | async">
        <a [routerLink]="['/bars', bar.id]">
          {{bar.name}}
        </a>
      </li>
    </ul>
    <div>
      <router-outlet></router-outlet>
    </div>
  `,
  // Add our router directives we will be using
  directives: [ROUTER_DIRECTIVES]
})
export class AppComponent { }

```

RouterLink takes an array where first element is the path for routing and subsequent elements are for the dynamic routing parameters.

Child Routes

Sometimes it makes sense to nest view's or routes within one another. For example on the dashboard you want several sub views, similar to tabs but implemented via the routing system, to show the users' projects, contacts, messages ets. In order to support such scenarios the router allows us to define child routes.

First we adjust our RouterConfig from above and add the child routes:

```

import { ProjectsComponent } from '../components/projects.component';
import { MessagesComponent } from '../components/messages.component';

export const appRoutes: RouterConfig = [
  { path: '', pathMatch: 'full', redirectTo: 'login' },
  { path: 'dashboard', component: DashboardComponent,
    children: [
      { path: '', redirectTo: 'projects', pathMatch: 'full' },
      { path: 'projects', component: 'ProjectsComponent' },
      { path: 'messages', component: 'MessagesComponent' }
    ] },
  { path: 'bars/:id', component: BarDetailComponent },
  { path: 'login', component: LoginComponent },

```

```
{ path: 'signup', component: SignupComponent }
];
```

Now that we have our child routes defined we have to make sure those child routes can be displayed within our `DashboardComponent`, since that's where we have added the childs to. Previously we have learned that the components are displayed in a `<router-outlet></router-outlet>` tag. Similar we declare another `RouterOutlet` in the `DashboardComponent`:

```
import { Component } from '@angular/core';

@Component({
  selector: 'dashboard',
  template: `
    <a [routerLink]="['projects']">Projects</a>
    <a [routerLink]="['messages']">Messages</a>
    <div>
      <router-outlet></router-outlet>
    </div>
  `
})
export class DashboardComponent { }
```

As you can see, we have added another `RouterOutlet` in which the child routes will be displayed. Usually the route with an empty path will be shown, however, we set up a redirect to the `projects` route, because we want that to be shown immediately when the `dashboard` route is loaded. That being said, we need an empty route, otherwise you'll get an error like this:

```
Cannot match any routes: 'dashboard'
```

So by adding the *empty* route, meaning a route with an empty path, we have defined an entry point for the router.

ResolveData

This example will show you how you can resolve data fetched from a service before rendering your application's view.

Uses angular/router 3.0.0-beta.2 at the time of writing

users.service.ts

```
...
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Rx';
import { User } from './user.ts';

@Injectable()
export class UsersService {

  constructor(public http:Http) {}
```

```

/**
 * Returns all users
 * @returns {Observable<User[]>}
 */
index():Observable<User[]> {

    return this.http.get('http://mywebsite.com/api/v1/users')
        .map((res:Response) => res.json());
}

/**
 * Returns a user by ID
 * @param id
 * @returns {Observable<User>}
 */
get(id:number|string):Observable<User> {

    return this.http.get('http://mywebsite.com/api/v1/users/' + id)
        .map((res:Response) => res.json());
}
}

```

users.resolver.ts

```

...
import { UsersService } from './users.service.ts';
import { Observable } from 'rxjs/Rx';
import {
    Resolve,
    ActivatedRouteSnapshot,
    RouterStateSnapshot
} from "@angular/router";

@Injectable()
export class UsersResolver implements Resolve<User[] | User> {

    // Inject UsersService into the resolver
    constructor(private service:UsersService) {}

    resolve(route:ActivatedRouteSnapshot, state:RouterStateSnapshot):Observable<User[] | User>
    {
        // If userId param exists in current URL, return a single user, else return all users
        // Uses brackets notation to access `id` to suppress editor warning, may use dot
        // notation if you create an interface extending ActivatedRoute with an optional id? attribute
        if (route.params['id']) return this.service.get(route.params['id']);
        return this.service.index();
    }
}

```

users.component.ts

This is a page component with a list of all users. It will work similarly for User detail page component, replace `data.users` with `data.user` or whatever key defined in `app.routes.ts`(see below)

```
...
```

```

import { ActivatedRoute } from "@angular/router";

@Component(...)
export class UsersComponent {

  users:User[];

  constructor(route: ActivatedRoute) {
    route.data.subscribe(data => {
      // data['Match key defined in RouterConfig, see below']
      this.users = data.users;
    });
  }

  /**
   * It is not required to unsubscribe from the resolver as Angular's HTTP
   * automatically completes the subscription when data is received from the server
   */
}

```

app.routes.ts

```

...
import { UsersResolver } from './resolvers/users.resolver';

export const routes:RouterConfig = <RouterConfig>[
  ...
  {
    path: 'user/:id',
    component: UserComponent,
    resolve: {
      // hence data.user in UserComponent
      user: UsersResolver
    }
  },
  {
    path: 'users',
    component: UsersComponent,
    resolve: {
      // hence data.users in UsersComponent, note the pluralisation
      users: UsersResolver
    }
  },
  ...
]
...

```

app.resolver.ts

Optionally bundle multiple resolvers together.

IMPORTANT: *Services used in resolver must be imported first or you will get a 'No provider for ..Resolver error'. Remember that these services will be available globally and you will not need to*

declare them in any component's `providers` anymore. Be sure to unsubscribe from any subscription to prevent memory leak

```
...
import { UsersService } from './users.service';
import { UsersResolver } from './users.resolver';

export const ROUTE_RESOLVERS = [
  ...,
  UsersService,
  UsersResolver
]
```

main.browser.ts

Resolvers have to be injected during bootstrapping.

```
...
import {bootstrap} from '@angular/platform-browser-dynamic';
import { ROUTE_RESOLVERS } from './app.resolver';

bootstrap(<Type>App, [
  ...,
  ...ROUTE_RESOLVERS
])
.catch(err => console.error(err));
```

Routing with Children

Contrary to original documentation, I found this to be the way to properly nest children routes inside the `app.routing.ts` or `app.module.ts` file (depending on your preference). This approach works when using either WebPack or SystemJS.

The example below shows routes for `home`, `home/counter`, and `home/counter/fetch-data`. The first and last routes being examples of redirects. Finally at the end of the example is a proper way to export the Route to be imported in a separate file. For ex. `app.module.ts`

To further explain, Angular requires that you have a pathless route in the children array that includes the parent component, to represent the parent route. It's a little confusing but if you think about a blank URL for a child route, it would essentially equal the same URL as the parent route.

```
import { NgModule } from "@angular/core";
import { RouterModule, Routes } from "@angular/router";

import { HomeComponent } from "../components/home/home.component";
import { FetchDataComponent } from "../components/fetchdata/fetchdata.component";
import { CounterComponent } from "../components/counter/counter.component";

const appRoutes: Routes = [
  {
    path: "",
    redirectTo: "home",
    pathMatch: "full"
  },
  {
    path: "home",
    component: HomeComponent,
    children: [
      {
        path: "counter",
        component: CounterComponent,
        children: [
          {
            path: "fetch-data",
            component: FetchDataComponent
          }
        ]
      }
    ]
  }
]
```

```

    {
      path: "home",
      children: [
        {
          path: "",
          component: HomeComponent
        },
        {
          path: "counter",
          children: [
            {
              path: "",
              component: CounterComponent
            },
            {
              path: "fetch-data",
              component: FetchDataComponent
            }
          ]
        }
      ]
    },
    {
      path: "**",
      redirectTo: "home"
    }
  ];

  @NgModule({
    imports: [
      RouterModule.forRoot(appRoutes)
    ],
    exports: [
      RouterModule
    ]
  })
  export class AppRoutingModule { }

```

[Great Example and Description via Siraj](#)

[Read Routing online: https://riptutorial.com/angular2/topic/2334/routing](https://riptutorial.com/angular2/topic/2334/routing)

Chapter 60: Routing (3.0.0+)

Remarks

There are a few more tricks we can do with the router (such as restricting access), but those can be covered in a separate tutorial.

If you need a new route, simply modify `app.routes.ts` and follow the following steps:

1. Import the Component
2. Add to the `routes` array. Make sure to include a new `path` and `component`.

Examples

Bootstrapping

Now that the routes are defined, we need to let our application know about the routes. To do this, bootstrap the provider we exported in the previous example.

Find your bootstrap configuration (should be in `main.ts`, but **your mileage may vary**).

```
//main.ts

import {bootstrap} from '@angular/platform-browser-dynamic';

//Import the App component (root component)
import { App } from './app/app';

//Also import the app routes
import { APP_ROUTES_PROVIDER } from './app/app.routes';

bootstrap(App, [
  APP_ROUTES_PROVIDER,
])
.catch(err => console.error(err));
```

Configuring router-outlet

Now that the router is configured and our app knows how to handle the routes, we need to show the actual components that we configured.

To do so, configure your HTML template/file for your **top-level (app)** component like so:

```
//app.ts

import {Component} from '@angular/core';
import {Router, ROUTER_DIRECTIVES} from '@angular/router';

@Component({
  selector: 'app',
```



```

    templateUrl: 'app.html',
    styleUrls: ['app.css'],
    directives: [
        ROUTER_DIRECTIVES,
    ]
})
export class App {
    constructor() {
    }
}

<!-- app.html -->

<!-- All of your 'views' will go here -->
<router-outlet></router-outlet>

```

The `<router-outlet></router-outlet>` element will switch the content given the route. Another good aspect about this element is that it *does not* have to be the only element in your HTML.

For example: Lets say you wanted a a toolbar on every page that stays constant between routes, similar to how Stack Overflow looks. You can nest the `<router-outlet>` under elements so that only certain parts of the page change.

Changing routes (using templates & directives)

Now that the routes are set up, we need some way to actually change routes.

This example will show how to change routes using the template, but it is possible to change routes in TypeScript.

Here is one example (without binding):

```
<a routerLink="/home">Home</a>
```

If the user clicks on that link, it will route to `/home`. The router knows how to handle `/home`, so it will display the `Home` Component.

Here is an example with data binding:

```
<a *ngFor="let link of links" [routerLink]="link">{{link}}</a>
```

Which would require an array called `links` to exist, so add this to `app.ts`:

```

public links[] = [
    'home',
    'login'
]

```

This will loop through the array and add an `<a>` element with the `routerLink` directive = the value of the current element in the array, creating this:

```
<a routerLink="home">home</a>
<a routerLink="login">login</a>
```

This is particularly helpful if you have a lot of links, or maybe the links need to be constantly changed. We let Angular handle the busy work of adding links by just feeding it the info it requires.

Right now, `links[]` is static, but it is possible to feed it data from another source.

Setting the Routes

NOTE: This example is based on the 3.0.0.-beta.2 release of the `@angular/router`. At the time of writing, this is the latest version of the router.

To use the router, define routes in a new TypeScript file like such

```
//app.routes.ts

import {provideRouter} from '@angular/router';

import {Home} from './routes/home/home';
import {Profile} from './routes/profile/profile';

export const routes = [
  {path: '', redirectTo: 'home'},
  {path: 'home', component: Home},
  {path: 'login', component: Login},
];

export const APP_ROUTES_PROVIDER = provideRouter(routes);
```

In the first line, we import `provideRouter` so we can let our application know what the routes are during the bootstrap phase.

`Home` and `Profile` are just two components as an example. You will need to import each `Component` you need as a route.

Then, export the array of routes.

`path`: The path to the component. **YOU DO NOT NEED TO USE `'/.....'`** Angular will do this automatically

`component`: The component to load when the route is accessed

`redirectTo`: *Optional*. If you need to automatically redirect a user when they access a particular route, supply this.

Finally, we export the configured router. `provideRouter` will return a provider that we can bootstrap so our application knows how to handle each route.

Controlling Access to or from a Route

The default Angular router allows navigation to and from any route unconditionally. This is not

always the desired behavior.

In a scenario where a user may conditionally be allowed to navigate to or from a route, a **Route Guard** may be used to restrict this behavior.

If your scenario fits one of the following, consider using a Route Guard,

- User is required to be authenticated to navigate to the target component.
- User is required to be authorized to navigate to the target component.
- Component requires asynchronous request before initialization.
- Component requires user input before navigated away from.

How Route Guards work

Route Guards work by returning a boolean value to control the behavior of router navigation. If *true* is returned, the router will continue with navigation to the target component. If *false* is returned, the router will deny navigation to the target component.

Route Guard Interfaces

The router supports multiple guard interfaces:

- *CanActivate*: occurs between route navigation.
- *CanActivateChild*: occurs between route navigation to a child route.
- *CanDeactivate*: occurs when navigating away from the current route.
- *CanLoad*: occurs between route navigation to a feature module loaded asynchronously.
- *Resolve*: used to perform data retrieval before route activation.

These interfaces can be implemented in your guard to grant or remove access to certain processes of the navigation.

Synchronous vs. Asynchronous Route Guards

Route Guards allow synchronous and asynchronous operations to conditionally control navigation.

Synchronous Route Guard

A synchronous route guard returns a boolean, such as by computing an immediate result, in order

to conditionally control navigation.

```
import { Injectable }    from '@angular/core';
import { CanActivate }  from '@angular/router';

@Injectable()
export class SynchronousGuard implements CanActivate {
  canActivate() {
    console.log('SynchronousGuard#canActivate called');
    return true;
  }
}
```

Asynchronous Route Guard

For more complex behavior, a route guard can asynchronously block navigation. An asynchronous route guard can return an Observable or Promise.

This is useful for situations like waiting for user input to answer a question, waiting to successfully save changes to the server, or waiting to receive data fetched from a remote server.

```
import { Injectable }    from '@angular/core';
import { CanActivate, Router, ActivatedRouteSnapshot, RouterStateSnapshot } from
 '@angular/router';
import { Observable }    from 'rxjs/Rx';
import { MockAuthenticationService } from './authentication/authentication.service';

@Injectable()
export class AsynchronousGuard implements CanActivate {
  constructor(private router: Router, private auth: MockAuthenticationService) {}

  canActivate(route:ActivatedRouteSnapshot,
state:RouterStateSnapshot):Observable<boolean>|boolean {
    this.auth.subscribe((authenticated) => {
      if (authenticated) {
        return true;
      }
      this.router.navigateByUrl('/login');
      return false;
    });
  }
}
```

Add guard to route configuration

File *app.routes*

Protected routes have `canActivate` binded to `Guard`

```
import { provideRouter, Router, RouterConfig, CanActivate } from '@angular/router';

//components
import { LoginComponent } from './login/login.component';
```

```
import { DashboardComponent } from './dashboard/dashboard.component';

export const routes: RouterConfig = [
  { path: 'login', component: LoginComponent },
  { path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] }
]
```

Export the **APP_ROUTER_PROVIDERS** to be used in app bootstrap

```
export const APP_ROUTER_PROVIDERS = [
  AuthGuard,
  provideRouter(routes)
];
```

Use Guard in app bootstrap

File *main.ts* (or *boot.ts*)

Consider the examples above:

1. **Create the guard** (where the Guard is created) and
2. **Add guard to route configuration**, (where the Guard is configured for route, then **APP_ROUTER_PROVIDERS** is exported),
we can couple the bootstrap to Guard as follows

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import { provide } from '@angular/core';

import { APP_ROUTER_PROVIDERS } from './app.routes';
import { AppComponent } from './app.component';

bootstrap(AppComponent, [
  APP_ROUTER_PROVIDERS
])
.then(success => console.log(`Bootstrap success`))
.catch(error => console.log(error));
```

Using Resolvers and Guards

We're using a toplevel guard in our route config to catch the current user on first page load, and a resolver to store the value of the `currentUser`, which is our authenticated user from the backend.

A simplified version of our implementation looks as follows:

Here is our top level route:

```
export const routes = [
  {
    path: 'Dash',
    pathMatch: 'prefix',
    component: DashCmp,
    canActivate: [AuthGuard],
    resolve: {
```

```

        currentUser: CurrentUserResolver
    },
    children: [...[
        path: '',
        component: ProfileCmp,
        resolve: {
            currentUser: currentUser
        }
    ]]
    }
];

```

Here is our AuthService

```

import { Injectable } from '@angular/core';
import { Http, Headers, RequestOptions } from '@angular/http';
import { Observable } from 'rxjs/Rx';
import 'rxjs/add/operator/do';

@Injectable()
export class AuthService {
    constructor(http: Http) {
        this.http = http;

        let headers = new Headers({ 'Content-Type': 'application/json' });
        this.options = new RequestOptions({ headers: headers });
    }
    fetchCurrentUser() {
        return this.http.get('/api/users/me')
            .map(res => res.json())
            .do(val => this.currentUser = val);
    }
}

```

Here is our AuthGuard:

```

import { Injectable } from '@angular/core';
import { CanActivate } from "@angular/router";
import { Observable } from 'rxjs/Rx';

import { AuthService } from '../services/AuthService';

@Injectable()
export class AuthGuard implements CanActivate {
    constructor(auth: AuthService) {
        this.auth = auth;
    }
    canActivate(route, state) {
        return Observable
            .merge(this.auth.fetchCurrentUser(), Observable.of(true))
            .filter(x => x == true);
    }
}

```

Here is our CurrentUserResolver:

```

import { Injectable } from '@angular/core';

```

```
import { Resolve } from "@angular/router";
import { Observable } from 'rxjs/Rx';

import { AuthService } from '../services/AuthService';

@Injectable()
export class CurrentUserResolver implements Resolve {
  constructor(auth: AuthService) {
    this.auth = auth;
  }
  resolve(route, state) {
    return this.auth.currentUser;
  }
}
```

Read Routing (3.0.0+) online: <https://riptutorial.com/angular2/topic/1208/routing--3-0-0plus->

Chapter 61: Service Worker

Introduction

We will see how to set up a service working on angular, to allow our web app to have offline capabilities.

A Service worker is a special script which runs in the background in the browser and manages network requests to a given origin. It's originally installed by an app and stays resident on the user's machine/device. It's activated by the browser when a page from its origin is loaded and has the option to respond to HTTP requests during the page loading

Examples

Add Service Worker to our app

First in case you are consulting mobile.angular.io the flag --mobile doesn't work anymore.

So to start , we can create a normal project with angular cli.

```
ng new serviceWorking-example
cd serviceWorking-example
```

Now the important thing, to said to angular cli that we want to use service worker we need to do:

```
ng set apps.0.serviceWorker=true
```

If for some reason you don't have @angular/service-worker installed, you will see a message:

```
Your project is configured with serviceWorker = true, but @angular/service-worker is
not installed. Run npm install --save-dev @angular/service-worker and try again, or run
ng set apps.0.serviceWorker=false in your .angular-cli.json.
```

Check the .angular-cli.json and you now should see this: "serviceWorker": true

When this flag is true, production builds will be set up with a service worker.

A ngsw-manifest.json file will be generated (or augmented in case we have create a ngsw-manifest.json in the root of the project, usually this is done to specify the routing ,in a future this will probably be done automatic) in the dist/ root, and the service worker script will be copied there. A short script will be added to index.html to register the service worker.

Now if we build the app in production mode ng build --prod

And check dist/ folder.

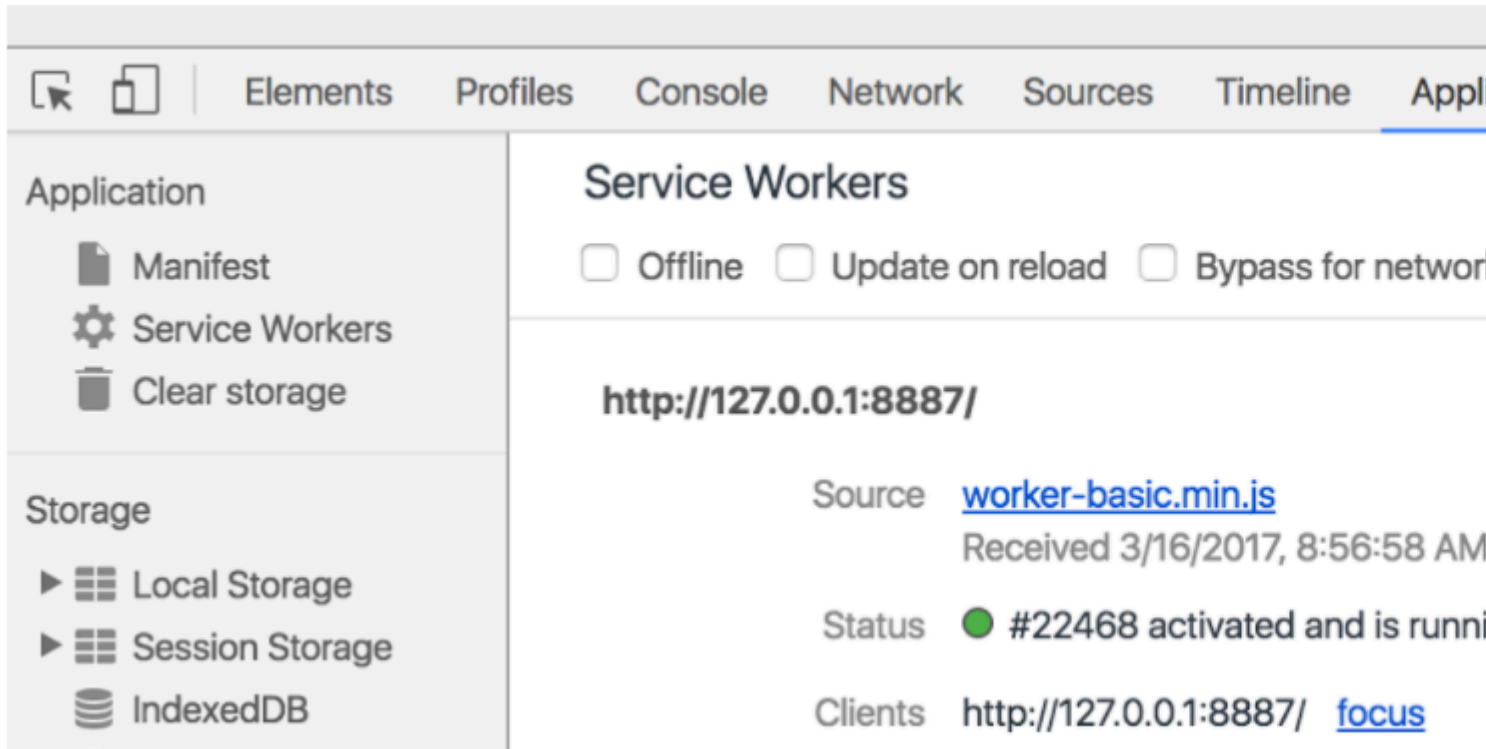
You will see three new files there :

- worker-basic.min.js
- sw-register.HASH.bundle.js
- ngsw-manifest.json

Also, index.html now includes this sw-register script, which registers a Angular Service Worker (ASW) for us.

Refresh the page in your browser (served by the Web Server for Chrome)

Open Developer Tools. Go to the Application -> Service Workers



Good now the Service Worker is up and running!

Now our application, should load faster and we should be able to use the app offline.

Now if you enable the offline mode in the chrome console , you should see that our app in <http://localhost:4200/index.html> is working without connection to internet.

But in <http://localhost:4200/> we have a problem and it doesn't load, this is due to the static content cache only serves files listed in the manifest.

For example, if the manifest declares a URL of /index.html, requests to /index.html will be answered by the cache, but a request to / or /some/route will go to the network.

That's where the route redirection plugin comes in. It reads a routing config from the manifest and redirects configured routes to a specified index route.

Currently, this section of configuration must be hand-written (19-7-2017). Eventually, it will be generated from the route configuration present in the application source.

So if now we create or `ngsw-manifest.json` in the root of the project

```
{
  "routing": {
    "routes": {
      "/": {
        "prefix": false
      }
    },
    "index": "/index.html"
  }
}
```

And we build again our app, now when we go to <http://localhost:4200/>, we should be redirected to <http://localhost:4200/index.html>.

For further information about routing read the [official documentation here](#)

Here you can find more documentation about service workers:

<https://developers.google.com/web/fundamentals/getting-started/primers/service-workers>

https://docs.google.com/document/d/19S5ozevWighny788nI99worpcIMDnwWVmaJDGf_RoDY/edit#

And here you can see an alternative way to implement the service working using SW precache library :

<https://coryryan.com/blog/fast-offline-angular-apps-with-service-workers>

Read Service Worker online: <https://riptutorial.com/angular2/topic/10809/service-worker>

Chapter 62: Services and Dependency Injection

Examples

Example service

services/my.service.ts

```
import { Injectable } from '@angular/core';

@Injectable()
export class MyService {
  data: any = [1, 2, 3];

  getData() {
    return this.data;
  }
}
```

The service provider registration in the bootstrap method will make the service available globally.

main.ts

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import { AppComponent } from 'app.component.ts';
import { MyService } from 'services/my.service';

bootstrap(AppComponent, [MyService]);
```

In version RC5 global service provider registration can be done inside the module file. In order to get a single instance of your service for your whole application the service should be declared in the providers list in the ngmodule of your application. *app_module.ts*

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { routing, appRoutingProviders } from './app-routes/app.routes';
import { HttpClientModule } from '@angular/http';

import { AppComponent } from './app.component';
import { MyService } from 'services/my.service';

import { routing } from './app-resources/app-routes/app.routes';

@NgModule({
  declarations: [ AppComponent ],
  imports: [ BrowserModule,
            routing,
            RouterModule,
            HttpClientModule ],
  providers: [ appRoutingProviders,
```

```

        MyService
    ],
    bootstrap: [AppComponent],
  })
  export class AppModule {}

```

Usage in `MyComponent`

components/my.component.ts

Alternative approach to register application providers in application components. If we add providers at component level whenever the component is rendered it will create a new instance of the service.

```

import { Component, OnInit } from '@angular/core';
import { MyService } from '../services/my.service';

@Component({
  ...
  providers:[MyService] //
})
export class MyComponent implements OnInit {
  data: any[];
  // Creates private variable myService to use, of type MyService
  constructor(private myService: MyService) { }

  ngOnInit() {
    this.data = this.myService.getData();
  }
}

```

Example with `Promise.resolve`

services/my.service.ts

```

import { Injectable } from '@angular/core';

@Injectable()
export class MyService {
  data: any = [1, 2, 3];

  getData() {
    return Promise.resolve(this.data);
  }
}

```

`getData()` now acts like a REST call that creates a Promise, which gets resolved immediately. The results can be handheld inside `.then()` and errors can also be detected. This is good practice and convention for asynchronous methods.

components/my.component.ts

```

import { Component, OnInit } from '@angular/core';

```

```

import { MyService } from '../services/my.service';

@Component({...})
export class MyComponent implements OnInit {
  data: any[];
  // Creates private variable myService to use, of type MyService
  constructor(private myService: MyService) { }

  ngOnInit() {
    // Uses an "arrow" function to set data
    this.myService.getData().then(data => this.data = data);
  }
}

```

Testing a Service

Given a service that can login a user:

```

import 'rxjs/add/operator/toPromise';

import { Http } from '@angular/http';
import { Injectable } from '@angular/core';

interface LoginCredentials {
  password: string;
  user: string;
}

@Injectable()
export class AuthService {
  constructor(private http: Http) { }

  async signIn({ user, password }: LoginCredentials) {
    const response = await this.http.post('/login', {
      password,
      user,
    }).toPromise();

    return response.json();
  }
}

```

It can be tested like this:

```

import { ConnectionBackend, Http, HttpClientModule, Response, ResponseOptions } from
 '@angular/http';
import { TestBed, async, inject } from '@angular/core/testing';

import { AuthService } from './auth.service';
import { MockBackend } from '@angular/http/testing';
import { MockConnection } from '@angular/http/testing';

describe('AuthService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientModule],
      providers: [
        AuthService,

```

```

    Http,
    { provide: ConnectionBackend, useClass: MockBackend },
  ]
});
});

it('should be created', inject([AuthService], (service: AuthService) => {
  expect(service).toBeTruthy();
}));

// Alternative 1
it('should login user if right credentials are passed', async(
  inject([AuthService], async (authService) => {
    const backend: MockBackend = TestBed.get(ConnectionBackend);
    const http: Http = TestBed.get(Http);

    backend.connections.subscribe((c: MockConnection) => {
      c.mockRespond(
        new Response(
          new ResponseOptions({
            body: {
              accessToken: 'abcdef',
            },
          }),
        ),
      );
    });

    const result = await authService.signIn({ password: 'ok', user: 'bruno' });

    expect(result).toEqual({
      accessToken: 'abcdef',
    });
  })
);

// Alternative 2
it('should login user if right credentials are passed', async () => {
  const backend: MockBackend = TestBed.get(ConnectionBackend);
  const http: Http = TestBed.get(Http);

  backend.connections.subscribe((c: MockConnection) => {
    c.mockRespond(
      new Response(
        new ResponseOptions({
          body: {
            accessToken: 'abcdef',
          },
        }),
      ),
    );
  });

  const authService: AuthService = TestBed.get(AuthService);

  const result = await authService.signIn({ password: 'ok', user: 'bruno' });

  expect(result).toEqual({
    accessToken: 'abcdef',
  });
});

```

```

// Alternative 3
it('should login user if right credentials are passed', async (done) => {
  const authService: AuthService = TestBed.get(AuthService);

  const backend: MockBackend = TestBed.get(ConnectionBackend);
  const http: Http = TestBed.get(Http);

  backend.connections.subscribe((c: MockConnection) => {
    c.mockRespond(
      new Response(
        new ResponseOptions({
          body: {
            accessToken: 'abcdef',
          },
        }),
      ),
    );
  });

  try {
    const result = await authService.signIn({ password: 'ok', user: 'bruno' });

    expect(result).toEqual({
      accessToken: 'abcdef',
    });

    done();
  } catch (err) {
    fail(err);
    done();
  }
});
});

```

Read Services and Dependency Injection online:

<https://riptutorial.com/angular2/topic/4187/services-and-dependency-injection>

Chapter 63: Templates

Introduction

Templates are very similar to templates in Angular 1, though there are many small syntactical changes that make it more clear what is happening.

Examples

Angular 2 Templates

A SIMPLE TEMPLATE

Let's start with a very simple template that shows our name and our favorite thing:

```
<div>
  Hello my name is {{name}} and I like {{thing}} quite a lot.
</div>
```

{ }: RENDERING

To render a value, we can use the standard double-curly syntax:

```
My name is {{name}}
```

Pipes, previously known as “Filters,” transform a value into a new value, like localizing a string or converting a floating point value into a currency representation:

[]: BINDING PROPERTIES

To resolve and bind a variable to a component, use the [] syntax. If we have this.currentVolume in our component, we will pass this through to our component and the values will stay in sync:

```
<video-control [volume]="currentVolume"></video-control>
(): HANDLING EVENTS
```

() : HANDLING EVENTS To listen for an event on a component, we use the () syntax

```
<my-component (click)="onClick($event)"></my-component>
```

[()]: TWO-WAY DATA BINDING

To keep a binding up to date given user input and other events, use the [()] syntax. Think of it as a combination of handling an event and binding a property:

```
<input [(ngModel)]="myName"> The this.myName value of your component will stay in sync with
```


the input value.

*: THE ASTERISK

Indicates that this directive treats this component as a template and will not draw it as-is. For example, ngFor takes our and stamps it out for each item in items, but it never renders our initial since it's a template:

```
<my-component *ngFor="#item of items">  
</my-component>
```

Other similar directives that work on templates rather than rendered components are *ngIf and *ngSwitch.

Read Templates online: <https://riptutorial.com/angular2/topic/9471/templates>

Chapter 64: Testing an Angular 2 App

Examples

Installing the Jasmine testing framework

The most common way to test Angular 2 apps is with the Jasmine test framework. Jasmine allows you to test your code in the browser.

Install

To get started, all you need is the `jasmine-core` package (not `jasmine`).

```
npm install jasmine-core --save-dev --save-exact
```

Verify

To verify that Jasmine is set up properly, create the file `./src/unit-tests.html` with the following content and open it in the browser.

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8">
  <title>Ng App Unit Tests</title>
  <link rel="stylesheet" href="../node_modules/jasmine-core/lib/jasmine-core/jasmine.css">
  <script src="../node_modules/jasmine-core/lib/jasmine-core/jasmine.js"></script>
  <script src="../node_modules/jasmine-core/lib/jasmine-core/jasmine-html.js"></script>
  <script src="../node_modules/jasmine-core/lib/jasmine-core/boot.js"></script>
</head>
<body>
  <!-- Unit Testing Chapter #1: Proof of life. -->
  <script>
    it('true is true', function () {
      expect(true).toEqual(true);
    });
  </script>
</body>
</html>
```

Setting up testing with Gulp, Webpack, Karma and Jasmine

The first thing we need is to tell karma to use Webpack to read our tests, under a configuration we set for the webpack engine. Here, I am using babel because I write my code in ES6, you can change that for other flavors, such as Typescript. Or I use Pug (formerly Jade) templates, you don't have to.

Still, the strategy remains the same.

So, this is a webpack config:

```
const webpack = require("webpack");
let packConfig = {
  entry: {},
  output: {},
  plugins:[
    new webpack.DefinePlugin({
      ENVIRONMENT: JSON.stringify('test')
    })
  ],
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude:/(node_modules|bower_components)/,
        loader: "babel",
        query:{
          presets:["es2015", "angular2"]
        }
      },
      {
        test: /\.woff2??$|\.ttf?$|\.eot?$|\.svg$/,
        loader: "file"
      },
      {
        test: /\.scss$/,
        loaders: ["style", "css", "sass"]
      },
      {
        test: /\.pug$/,
        loader: 'pug-html-loader'
      },
    ],
    devtool : 'inline-cheap-source-map'
  };
module.exports = packConfig;
```

And then, we need a karma.config.js file to use that webpack config:

```
const packConfig = require("./webpack.config.js");
module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    exclude:[],
    files: [
      {pattern: './karma.shim.js', watched: false}
    ],

    preprocessors: {
      './karma.shim.js':['webpack']
    },
    webpack: packConfig,

    webpackServer: {noInfo: true},
```

```

    port: 9876,

    colors: true,

    logLevel: config.LOG_INFO,

    browsers: ['PhantomJS'],

    concurrency: Infinity,

    autoWatch: false,
    singleRun: true
  });
};

```

So far, we have told Karma to use webpack, and we have told it to start at a file called **karma.shim.js**. This file will have the job of acting as the starting point for webpack. webpack will read this file and use the **import** and **require** statements to gather all our dependencies and run our tests.

So now, let's look at the karma.shim.js file:

```

// Start of ES6 Specific stuff
import "es6-shim";
import "es6-promise";
import "reflect-metadata";
// End of ES6 Specific stuff

import "zone.js/dist/zone";
import "zone.js/dist/long-stack-trace-zone";
import "zone.js/dist/jasmine-patch";
import "zone.js/dist/async-test";
import "zone.js/dist/fake-async-test";
import "zone.js/dist/sync-test";
import "zone.js/dist/proxy-zone";

import 'rxjs/add/operator/map';
import 'rxjs/add/observable/of';

Error.stackTraceLimit = Infinity;

import {TestBed} from "@angular/core/testing";
import { BrowserDynamicTestingModule, platformBrowserDynamicTesting } from "@angular/platform-browser-dynamic/testing";

TestBed.initTestEnvironment (
  BrowserDynamicTestingModule,
  platformBrowserDynamicTesting());

let testContext = require.context('../src/app', true, /\.spec\.js/);
testContext.keys().forEach(testContext);

```

In essence, we are importing **TestBed** from angular core testing, and initiating the environment, as it needs to be initiated only once for all of our tests. Then, we are going through the **src/app** directory recursively and reading every file that ends with **.spec.js** and feed them to testContext, so they will run.

I usually try to put my tests the same place as the class. Personal taste, it makes it easier for me to import dependencies and refactor tests with classes. But if you want to put your tests somewhere else, like under **src/test** directory for example, here is your chance. change the line before last in the karma.shim.js file.

Perfect. what is left? ah, the gulp task that uses the karma.config.js file we made above:

```
gulp.task("karmaTests", function(done) {
  var Server = require("karma").Server;
  new Server({
    configFile : "./karma.config.js",
    singleRun: true,
    autoWatch: false
  }, function(result){
    return result ? done(new Error(`Karma failed with error code ${result}`)):done();
  }).start();
});
```

I am now starting the server with the config file we created, telling it to run once and don't watch for changes. I find this to suite me better as the tests will run only if I am ready for them to run, but of course if you want different you know where to change.

And as my final code sample, here is a set of tests for the Angular 2 tutorial, "Tour of Heroes".

```
import {
  TestBed,
  ComponentFixture,
  async
} from "@angular/core/testing";

import {AppComponent} from "../app.component";
import {AppModule} from "../app.module";
import Hero from "../hero/hero";

describe("App Component", function () {

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [AppModule]
    });

    this.fixture = TestBed.createComponent(AppComponent);
    this.fixture.detectChanges();
  });

  it("Should have a title", async(() => {
    this.fixture.whenStable().then(() => {
      expect(this.fixture.componentInstance.title).toEqual("Tour of Heros");
    });
  }));

  it("Should have a hero", async(() => {
    this.fixture.whenStable().then(() => {
      expect(this.fixture.componentInstance.selectedHero).toBeNull();
    });
  }));
});
```

```

it("Should have an array of heros", async(()=>
  this.fixture.whenStable().then(()=> {
    const cmp = this.fixture.componentInstance;
    expect(cmp.heroes).toBeDefined("component should have a list of heroes");
    expect(cmp.heroes.length).toEqual(10, "heroes list should have 10 members");
    cmp.heroes.map((h, i)=> {
      expect(h instanceof Hero).toBeTruthy(`member ${i} is not a Hero instance.
${h}`)
    });
  }));

it("Should have one list item per hero", async(()=>
  this.fixture.whenStable().then(()=> {
    const ul = this.fixture.nativeElement.querySelector("ul.heroes");
    const li = Array.prototype.slice.call(
      this.fixture.nativeElement.querySelectorAll("ul.heroes>li"));
    const cmp = this.fixture.componentInstance;
    expect(ul).toBeTruthy("There should be an unnumbered list for heroes");
    expect(li.length).toEqual(cmp.heroes.length, "there should be one li for each
hero");
    li.forEach((li, i)=> {
      expect(li.querySelector("span.badge"))
        .toBeTruthy(`hero ${i} has to have a span for id`);
      expect(li.querySelector("span.badge").textContent.trim())
        .toEqual(cmp.heroes[i].id.toString(), `hero ${i} had wrong id displayed`);
      expect(li.textContent)
        .toMatch(cmp.heroes[i].name, `hero ${i} has wrong name displayed`);
    });
  }));

it("should have correct styling of hero items", async(()=>
  this.fixture.whenStable().then(()=> {
    const hero = this.fixture.nativeElement.querySelector("ul.heroes>li");
    const win = hero.ownerDocument.defaultView || hero.ownerDocument.parentWindow;
    const styles = win.getComputedStyle(hero);
    expect(styles["cursor"]).toEqual("pointer", "cursor should be pointer on hero");
    expect(styles["borderRadius"]).toEqual("4px", "borderRadius should be 4px");
  }));

it("should have a click handler for hero items", async(()=>
  this.fixture.whenStable().then(()=>{
    const cmp = this.fixture.componentInstance;
    expect(cmp.onSelect)
      .toBeDefined("should have a click handler for heros");
    expect(this.fixture.nativeElement.querySelector("input.heroName"))
      .toBeNull("should not show the hero details when no hero has been selected");
    expect(this.fixture.nativeElement.querySelector("ul.heroes li.selected"))
      .toBeNull("Should not have any selected heroes at start");

    spyOn(cmp, "onSelect").and.callThrough();
    this.fixture.nativeElement.querySelectorAll("ul.heroes li")[5].click();

    expect(cmp.onSelect)
      .toHaveBeenCalledWith(cmp.heroes[5]);
    expect(cmp.selectedHero)
      .toEqual(cmp.heroes[5], "click on hero should change hero");
  })
));
});

```

Noteworthy in this is how we have **beforeEach()** configure a test module and create the

component in test, and how we call **detectChanges()** so that angular actually goes through the double-binding and all.

Notice that each test is a call to **async()** and it always waits for **whenStable** promise to resolve before examining the fixture. It then has access to the component through **componentInstance** and to the element through **nativeElement**.

There is one test which is checking the correct styling. as part of the Tutorial, Angular team demonstrates use of styles inside components. In our test, we use **getComputedStyle()** to check that styles are coming from where we specified, however we need the Window object for that, and we are getting it from the element as you can see in the test.

Testing Http Service

Usually, services call remote Api to retrieve/send data. But unit tests shouldn't do network calls. Angular internally uses `XHRBackend` class to do http requests. User can override this to change behavior. Angular testing module provides `MockBackend` and `MockConnection` classes which can be used to test and assert http requests.

`posts.service.ts` This service hits an api endpoint to fetch list of posts.

```
import { Http } from '@angular/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/rx';

import 'rxjs/add/operator/map';

export interface IPost {
  userId: number;
  id: number;
  title: string;
  body: string;
}

@Injectable()
export class PostsService {
  posts: IPost[];

  private postsUri = 'http://jsonplaceholder.typicode.com/posts';

  constructor(private http: Http) {
  }

  get(): Observable<IPost[]> {
    return this.http.get(this.postsUri)
      .map((response) => response.json());
  }
}
```

`posts.service.spec.ts` Here, we will test above service by mocking http api calls.

```
import { TestBed, inject, fakeAsync } from '@angular/core/testing';
import {
  HttpClientModule,
```

```

    XHRBackend,
    ResponseOptions,
    Response,
    RequestMethod
} from '@angular/http';
import {
    MockBackend,
    MockConnection
} from '@angular/http/testing';

import { PostsService } from './posts.service';

describe('PostsService', () => {
    // Mock http response
    const mockResponse = [
        {
            'userId': 1,
            'id': 1,
            'title': 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit',
            'body': 'quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto'
        },
        {
            'userId': 1,
            'id': 2,
            'title': 'qui est esse',
            'body': 'est rerum tempore vitae\nsequi sint nihil reprehenderit dolor beatae ea dolores neque\nfugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis\nqui aperiam non debitis possimus qui neque nisi nulla'
        },
        {
            'userId': 1,
            'id': 3,
            'title': 'ea molestias quasi exercitationem repellat qui ipsa sit aut',
            'body': 'et iusto sed quo iure\nvoluptatem occaecati omnis eligendi aut ad\nvoluptatem doloribus vel accusantium quis pariatur\nmolestiae porro eius odio et labore et velit aut'
        },
        {
            'userId': 1,
            'id': 4,
            'title': 'eum et est occaecati',
            'body': 'ullam et saepe reiciendis voluptatem adipisci\nsit amet autem assumenda provident rerum culpa\nquis hic commodi nesciunt rem tenetur doloremque ipsam iure\nquis sunt voluptatem rerum illo velit'
        }
    ];

    beforeEach(() => {
        TestBed.configureTestingModule({
            imports: [HttpModule],
            providers: [
                {
                    provide: XHRBackend,
                    // This provides mocked XHR backend
                    useClass: MockBackend
                },
                PostsService
            ]
        });
    });

```



```

    });
  });

  it('should return posts retrieved from Api', fakeAsync(
    inject([XHRBackend, PostsService],
      (mockBackend, postsService) => {
        mockBackend.connections.subscribe(
          (connection: MockConnection) => {
            // Assert that service has requested correct url with expected method
            expect(connection.request.method).toBe(RequestMethod.Get);

            expect(connection.request.url).toBe('http://jsonplaceholder.typicode.com/posts');
            // Send mock response
            connection.mockRespond(new Response(new ResponseOptions({
              body: mockResponse
            })));
          });

          postsService.get()
            .subscribe((posts) => {
              expect(posts).toBe(mockResponse);
            });

        }));
  });
});

```

Testing Angular Components - Basic

The component code is given as below.

```

import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: '<h1>{{title}}</h1>'
})
export class MyAppComponent {
  title = 'welcome';
}

```

For angular testing, angular provide its testing utilities along with the testing framework which helps in writing the good test case in angular. Angular utilities can be imported from `@angular/core/testing`

```

import { ComponentFixture, TestBed } from '@angular/core/testing';
import { MyAppComponent } from './banner-inline.component';

describe('Tests for MyAppComponent', () => {

  let fixture: ComponentFixture<MyAppComponent>;
  let comp: MyAppComponent;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [
        MyAppComponent
      ]
    });
  });
});

```

```

    });
  });

  beforeEach(() => {

    fixture = TestBed.createComponent(MyAppComponent);
    comp = fixture.componentInstance;

  });

  it('should create the MyAppComponent', () => {

    expect(comp).toBeTruthy();

  });

});

```

In the above example, there is only one test case which explain the test case for component existence. In the above example angular testing utilities like `TestBed` and `ComponentFixture` are used.

`TestBed` is used to create the angular testing module and we configure this module with the `configureTestingModule` method to produce the module environment for the class we want to test. Testing module to be configured before the execution of every test case that's why we configure the testing module in the `beforeEach` function.

`createComponent` method of `TestBed` is used to create the instance of the component under test. `createComponent` return the `ComponentFixture`. The fixture provides access to the component instance itself.

Read Testing an Angular 2 App online: <https://riptutorial.com/angular2/topic/2329/testing-an-angular-2-app>

Chapter 65: Testing ngModel

Introduction

Is a example for how you can test a component in Angular2 that have a ngModel.

Examples

Basic test

```
import { BrowserModule } from '@angular/platform-browser';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';

import { Component, DebugElement } from '@angular/core';
import { dispatchEvent } from "@angular/platform-browser/testing/browser_util";
import { TestBed, ComponentFixture } from '@angular/core/testing';
import { By } from "@angular/platform-browser";

import { MyComponentModule } from 'ng2-my-component';
import { MyComponent } from './my-component';

describe('MyComponent:', () => {

  const template = `
    <div>
      <my-component type="text" [(ngModel)]="value" name="TestName" size="9" min="3" max="8"
placeholder="testPlaceholder" disabled=false required=false></my-component>
    </div>
  `;

  let fixture:any;
  let element:any;
  let context:any;

  beforeEach(() => {

    TestBed.configureTestingModule({
      declarations: [InlineEditorComponent],
      imports: [
        FormsModule,
        InlineEditorModule]
    });
    fixture = TestBed.overrideComponent(InlineEditorComponent, {
      set: {
        selector:"inline-editor-test",
        template: template
      }
    })
    .createComponent(InlineEditorComponent);
    context = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should change value of the component', () => {
```

```
let input = fixture.nativeElement.querySelector("input");
input.value = "Username";
dispatchEvent(input, 'input');
fixture.detectChanges();

fixture.whenStable().then(() => {
  //this button dispatch event for save the text in component.value
  fixture.nativeElement.querySelectorAll('button')[0].click();
  expect(context.value).toBe("Username");
});
});
});
```

Read Testing ngModel online: <https://riptutorial.com/angular2/topic/8693/testing-ngmodel>

Chapter 66: unit testing

Examples

Basic unit test

component file

```
@Component ({
  selector: 'example-test-component',
  template: '<div>
    <div>{{user.name}}</div>
    <div>{{user.fname}}</div>
    <div>{{user.email}}</div>
  </div>'
})

export class ExampleTestComponent implements OnInit {

  let user :User = null;
  ngOnInit(): void {
    this.user.name = 'name';
    this.user.fname= 'fname';
    this.user.email= 'email';
  }

}
```

Test file

```
describe('Example unit test component', () => {
  let component: ExampleTestComponent ;
  let fixture: ComponentFixture<ExampleTestComponent >;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ExampleTestComponent]
    }).compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(ExampleTestComponent );
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('ngOnInit should change user object values', () => {
    expect(component.user).toBeNull(); // check that user is null on initialize
    component.ngOnInit(); // run ngOnInit

    expect(component.user.name).toEqual('name');
    expect(component.user.fname).toEqual('fname');
```

```
    expect(component.user.email).toEqual('email');  
  });  
});
```

Read unit testing online: <https://riptutorial.com/angular2/topic/8955/unit-testing>

Chapter 67: Update typings

Examples

Update typings when: typings WARN deprecated

Warning message:

```
typings WARN deprecated 10/25/2016: "registry:dt/jasmine#2.5.0+20161003201800" is deprecated  
(updated, replaced or removed)
```

Update the reference with:

```
npm run typings -- install dt~jasmine --save --global
```

Replace [jasmine] for any library that is throwing warning

Read Update typings online: <https://riptutorial.com/angular2/topic/7814/update-typings>

Chapter 68: Use native webcomponents in Angular 2

Remarks

When you use a web component in your Angular 2 template, angular will try to find a component with a selector matching the custom tag of the web component - which it of course can't and will throw an error.

The solution is to import a "custom elements schema" in the module that holds the component. This will make angular accept any custom tag, that doesn't match any ng component selector.

Examples

Include custom elements schema in your module

```
import { NgModule, CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';
import { CommonModule } from '@angular/common';
import { AboutComponent } from './about.component';

@NgModule({
  imports: [ CommonModule ],
  declarations: [ AboutComponent ],
  exports: [ AboutComponent ],
  schemas: [ CUSTOM_ELEMENTS_SCHEMA ]
})

export class AboutModule { }
```

Use your webcomponent in a template

```
import { Component } from '@angular/core';

@Component({
  selector: 'myapp-about',
  template: `<my-webcomponent></my-webcomponent>`
})
export class AboutComponent { }
```

Read Use native webcomponents in Angular 2 online:

<https://riptutorial.com/angular2/topic/7414/use-native-webcomponents-in-angular-2>

Chapter 69: Using third party libraries like jQuery in Angular 2

Introduction

When building applications using Angular 2.x there are times when it's required to use any third party libraries like jQuery, Google Analytics, Chat Integration JavaScript APIs and etc.

Examples

Configuration using angular-cli

NPM

If external library like `jQuery` is installed using NPM

```
npm install --save jquery
```

Add script path into your `angular-cli.json`

```
"scripts": [  
  "../node_modules/jquery/dist/jquery.js"  
]
```

Assets Folder

You can also save the library file in your `assets/js` directory and include the same in `angular-cli.json`

```
"scripts": [  
  "assets/js/jquery.js"  
]
```

Note

Save your main library `jquery` and their dependencies like `jquery-cycle-plugin` into the assets directory and add both of them into `angular-cli.json`, make sure the order is maintained for the dependencies.

Using jQuery in Angular 2.x components

To use `jquery` in your Angular 2.x components, declare a global variable on the top

If using `$` for jQuery

```
declare var $: any;
```

If using jQuery for jQuery

```
declare var jQuery: any
```

This will allow using `$` or `jQuery` into your Angular 2.x component.

Read [Using third party libraries like jQuery in Angular 2 online](https://riptutorial.com/angular2/topic/9285/using-third-party-libraries-like-jquery-in-angular-2):

<https://riptutorial.com/angular2/topic/9285/using-third-party-libraries-like-jquery-in-angular-2>

Chapter 70: Zone.js

Examples

Getting reference to NgZone

`NgZone` reference can be injected via the Dependency Injection (DI).

my.component.ts

```
import { Component, NgOnInit, NgZone } from '@angular/core';

@Component({...})
export class Mycomponent implements NgOnInit {
  constructor(private _ngZone: NgZone) { }
  ngOnInit() {
    this._ngZone.runOutsideAngular(() => {
      // Do something outside Angular so it won't get noticed
    });
  }
}
```

Using NgZone to do multiple HTTP requests before showing the data

`runOutsideAngular` can be used to run code outside Angular 2 so that it does not trigger change detection unnecessarily. This can be used to for example run multiple HTTP request to get all the data before rendering it. To execute code again inside Angular 2, `run` method of `NgZone` can be used.

my.component.ts

```
import { Component, OnInit, NgZone } from '@angular/core';
import { Http } from '@angular/http';

@Component({...})
export class Mycomponent implements OnInit {
  private data: any[];
  constructor(private http: Http, private _ngZone: NgZone) { }
  ngOnInit() {
    this._ngZone.runOutsideAngular(() => {
      this.http.get('resource1').subscribe((data1:any) => {
        // First response came back, so its data can be used in consecutive request
        this.http.get(`resource2?id=${data1['id']}`).subscribe((data2:any) => {
          this.http.get(`resource3?id1=${data1['id']}&id2=${data2}`).subscribe((data3:any) =>
          {
            this._ngZone.run(() => {
              this.data = [data1, data2, data3];
            });
          });
        });
      });
    });
  }
}
```

```
}  
}
```

Read Zone.js online: <https://riptutorial.com/angular2/topic/4184/zone-js>

Credits

S. No	Chapters	Contributors
1	Getting started with Angular 2	acdcjunior , Alexander Ciesielski , beagleknight , Bean0341 , Bhoomi Bhalani , BogdanC , briantylor , cDecker32 , Christopher Moore , Community , daniellmb , drbishop , echonax , Ekin Yücel , elliott-j , etayluz , ettanany , Everettss , H. Pauwelyn , Harry , He11ion , Janco Boscan , Jim , Kaspars Bergs , Logan H , Madhu Ranjan , michaelbahr , Michal Pietraszko , Mihai , nick , Nicolas Irisarri , Peter , QoP , rickysullivan , Shahzad , spike , theblindprophet , user6939352
2	Advanced Component Examples	borislemke , smnbbvr
3	Ahead-of-time (AOT) compilation with Angular 2	Anil Singh , Eric Jimenez , Harry , Robin Dijkhof
4	Angular 2 - Protractor	Yoav Schniederman
5	Angular - ForLoop	aholtry , Anil Singh , Berseker59 , gerl , Johan Van de Merwe , ob1 , Pujan Srivastava , Stephen Leppik , Yoav Schniederman
6	Angular 2 Change detection and manual triggering	Yoav Schniederman
7	Angular 2 Data Driven Forms	MatWaligora , ThunderRoid
8	Angular 2 Forms Update	Amit kumar , Anil Singh , Christopher Taylor , Highmastdon , Johan Van de Merwe , K3v1n , Manmeet Gill , mayur , Norsk , Sachin S , victoroniibukun , vijaykumar , Yoav Schniederman
9	Angular material design	Ketan Akbari , Shailesh Ladumor
10	angular redux	Yoav Schniederman
11	Angular RXJS Subjects and Observables with API requests	daniellmb , Maciej Treder , Ronald Zarīts , Sam Storie , Sébastien Temprado , willydee
12	Angular2 Animations	Yoav Schniederman
13	Angular2 CanActivate	Companjo , Yoav Schniederman

14	Angular2 Custom Validations	Arnold Wiersma , Norsk , Yoav Schniederman
15	Angular2 Databinding	Yoav Schniederman
16	Angular2 In Memory Web API	Jaime Still
17	Angular2 Input() output()	Kaloyan , Yoav Schniederman
18	Angular2 provide external data to App before bootstrap	Ajey
19	Angular2 using webpack	luukgruijs
20	Angular-cli	BogdanC , Yoav Schniederman
21	angular-cli test coverage	ahmadalibaloch
22	Animation	Gaurav Mukherjee , Nate May
23	Attribute directives to affect the value of properties on the host node by using the @HostBinding decorator.	Max Karpovets
24	Barrel	TechJhola
25	Bootstrap Empty module in angular 2	AryanJ-NYC , autoboxer , Berseker59 , Eric Jimenez , Krishan , Sanket , snorkpete
26	Brute Force Upgrading	Jim , Treveshan Naidoo
27	Bypassing Sanitizing for trusted values	Scrambo
28	Commonly built-in directives and services	Jim , Sanket
29	Component interactions	H. Pauwelyn , Janco Boscan , LLL , Sefa
30	Components	BrunoLM
31	Configuring ASP.net Core application to work with Angular 2 and TypeScript	Oleksii Aza , Sam
32	Create an Angular 2+ NPM package	BogdanC , Janco Boscan , vinagreti
33	Creating an Angular npm library	Maciej Treder
34	CRUD in Angular2 with Restful API	bleakgadfly , Sefa

35	custom ngx-bootstrap datepicker + input	Yoav Schniederman
36	Debugging Angular2 typescript application using Visual Studio Code	PSabuwala
37	Detecting resize events	Eric Jimenez
38	Directives	acdcjunior , Andrei Zhytkevich , borislemke , BrunoLM , daniellmb , Everettss , lexith , Stian Standahl , theblindprophet
39	Directives & components : @Input @Output	acdcjunior , dafyddPrys , Everettss , Joel Almeida , lexith , muetzerich , theblindprophet , ThomasP1988
40	Dropzone in Angular2	Ketan Akbari
41	Dynamically add components using ViewContainerRef.createComponent	amansoni211 , daniellmb , Günter Zöchbauer , jupiter24 , Khaled
42	EventEmitter Service	Abrar Jahin
43	Example for routes such as /route/subroute for static urls	Yoav Schniederman
44	Feature Modules	AryanJ-NYC , gsc
45	How to use ngfor	Jorge , Yoav Schniederman
46	How to Use ngif	Amit kumar , ob1 , ppovoski , samAlvin
47	Http Interceptor	Everettss , Mihai , Mike Kovetsky , Nilz11 , Paul Marshall , peeskillet , theblindprophet
48	Installing 3rd party plugins with angular-cli@1.0.0-beta.10	Alex Morales , Daredzik , filoxo , Kaspars Bergs , pd farhad
49	Lazy loading a module	Batajus , M4R1KU , Shannon Young , Syam Pradeep
50	Lifecycle Hooks	Alexandre Junges , daniellmb , Deen John , muetzerich , Sbats , theblindprophet
51	Mocking @ngrx/Store	BrianRT , Hatem , Jim , Lucas , Yoav Schniederman
52	Modules	BrunoLM
53	ngrx	Maxime

54	Optimizing rendering using ChangeDetectionStrategy	daniellmb , Eric Jimenez , Everettss
55	OrderBy Pipe	Yoav Schniederma
56	Page title	Yoav Schniederma
57	Pipes	acdcjunior , Boris , borislemke , BrunoLM , Christopher Taylor , Chybie , daniellmb , Daredzik , elliot-j , Everettss , Fredrik Lundin , Jarod Moser , Jeff Cross , Jim , Kaspars Bergs , Leon Adler , Lexi , LordTribual , michaelbahr , Philipp Kief , theblindprophet
58	Routing	aholtry , Apmis , AryanJ-NYC , borislemke , camwhite , Kaspars Bergs , LordTribual , Sachin S , theblindprophet
59	Routing (3.0.0+)	Ai_boy , Alexis Le Gal , Everettss , Gerard Simpson , Kaspars Bergs , mast3rd3mon , meorfi , rivanov , SlashTag , smnbbvr , theblindprophet , ThomasP1988 , Trent
60	Service Worker	Roberto Fernandez
61	Services and Dependency Injection	BrunoLM , Eduardo Carísio , Kaspars Bergs , Matrim , Roope Hakulinen , Syam Pradeep , theblindprophet
62	Templates	Max Karpovets
63	Testing an Angular 2 App	Arun Redhu , michaelbahr , nick , Reza , Rumit Parakhiya
64	Testing ngModel	jesussegado
65	unit testing	Yoav Schniederma
66	Update typings	kEpEx
67	Use native webcomponents in Angular 2	ugreen
68	Using third party libraries like jQuery in Angular 2	Ashok Vishwakarma
69	Zone.js	Roope Hakulinen